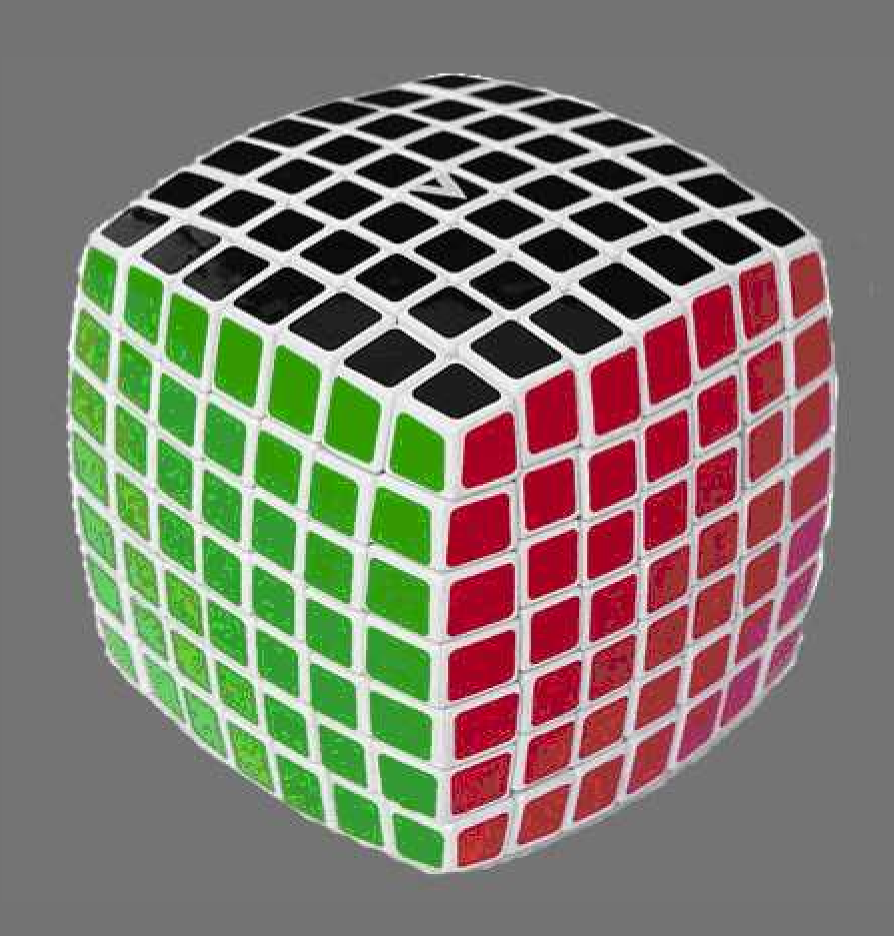
2010

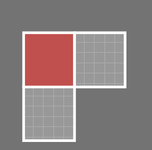
كيف تبرمج مترجما...

فهم برمجة الكومبايلر (Compiler)خطوة بخطوة



ياسين الجزائري

khatibe_30@hotmail.fr





الحمد لله رب العالمين والصلاة و السلام على المبعوث الأمين رحمة للعالمين محمد إبن عبد الله و على آله و صحبه و سلم تسليما كثيرا.

على الرغم من أن المكتبة العربية غنية جدا بعديد المصادر في شتى المواضيع لدرجة أن الطالب يضيع بين رفوف الكتب محاولا إيجاد الكتاب المناسب إلا أن الكتب التي تتناول موضوع الترجمة (La compilation) معدومة تقريبا, و في محاولة لتدعيم المكتبة العربية نقدم لكم هذا الكتاب الذي يشرح بطريقة سهلة و مفهومة الخطوات الأساسية لكيفية برمجة مترجم (Compilateur) خاص بك, لا يهم أن تكون مهمة هذا المترجم ترجمة أكواد لغة متقدمة إلى لغة الآلة بل يمكن إستعمال الكتاب لبرمجة برامج تحول الأكواد من لغة إلى أخرى, و لعل من أبرز الأمثلة البرامج المنتشرة التي تحول من اللغة C إلى PASCAL أو إلى JAVA و غيرها.

سنستهل كتابنا بمقدمة سريعة عن الترجمة و أصولها و من ثم عرض للأدوات اللازمة مع روابط تحميلها و كيفية تنصيبها, بعد ذلك نتطرق إلى مثال بسيط الهدف منه شرح الأدوات المستعملة في هذا الكتاب لنكمل بقية الكتاب في شرح كيفية برمجة مترجم خطوة بخطوة و سطرا بسطر.

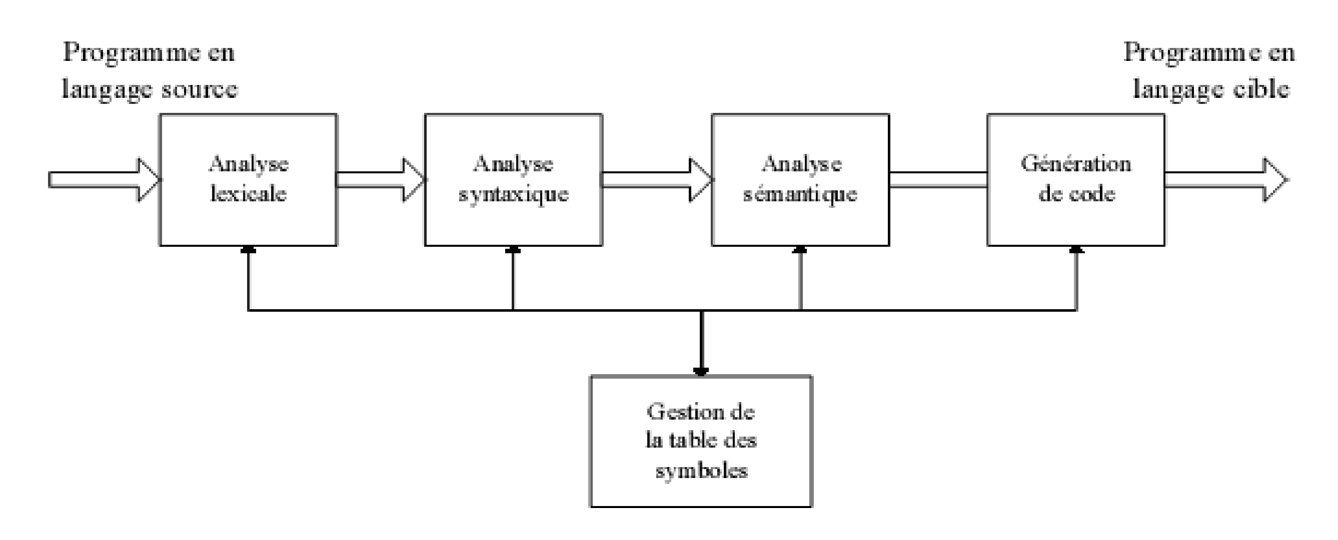
من الضروري جدا أن تحيط علما بأساسيات لغة البرمجة C++ لأنها ما سنعتمد عليه في هذا الكتاب بالإضافة إلى معرفة و إن كانت سطحية بلغة التجميع أو ما تعرف بالأسمبلي.

الفهرس

المترجم... هل فكرت يوما في برمجة مترجم ما كمترجم السي أو الباسكال أو غيرهما من اللغات؟ هل تعتقد أن الأمر صعب؟ أجل هو ليس باليسير و أيضا ليس بالمستحيل, و سنحاول في هذا الكتاب بإذن الله أن نوضح خطوات برمجة مترجم صغير.

1. مقدمة

لدينا برنامج مكتوب بلغة ما, إذا كانت هذه اللغة مفهومة من طرف الحاسوب فإنه يقوم بتنفيذ البرنامج مباشرة, أما إذا كانت لغة البرنامج غير مفهومة من طرف الحاسوب فيجب تحويل الكود المصدر (texte source) إلى كود مفهوم من طرف الحاسوب, (code cible), هذه العملية تسمى الترجمة, المترجم هو برنامج مكتوب بلغة ما يقوم بترجمة كود من لغة مصدر إلى لغة الآلة.



- مراحل ترجمة برنامج –

لنلقي نظرة سريعة على مراحل الترجمة.

1) L'analyse lexicale:

المرحلة الأولى من التحليل, يقوم المترجم هنا باستخراج الكلمات و التي تسمى tokens انطلاقا من سلسلة من الحروف, مثلا, لاحظ هذا السطر من الكود المصدر:

```
for i := 1 to vmax do a := a+i;
```

نستخرج هذه السلسلة من الـ tokens :

for	كلمة محجوزة(mot clé)
i	معرف (identificateur)
:=	إسناد (affectation)
1	صحيح (entier)
to	كلمة محجوزة (mot clé)
vmax	معرف (identificateur)
do	كلمة محجوزة (mot clé)
a	معرف (identificateur)
:=	إسناد (affectation)
a	معرف (identificateur)
+	عملية رياضية (opérateur arithmétique)
1	معرف (identificateur)
;	فاصل (séparateur)

مباشرة و بعد تحليل الكود نقوم ببناء جدول الرموز (table des symboles) وهو عبارة عن قائمة من مجموعة تركيبات تحمل خصائص كل token و يكون كالآتي:

Numéro de symbole	Token	Type de token	Type de variable
1	for	mot clé	...
2	to	mot clé	...
2	do	mot clé	...
3	;	séparateur	...
...	...	...	...

بهذه الطريقة يقوم المحلل المعجمي (L'analyseur lexicale) بتحليل الكود المصدر و بناء جدول الرموز –إن صحت ترجمة المصطلحات – و إن كنت قد لاحظت, فإن المحلل المعجمي لا يهتم بترتيب الـ tokens و لذلك فإن السطر التالي صحيح تماما بالنسبة للـ analyseur lexical:

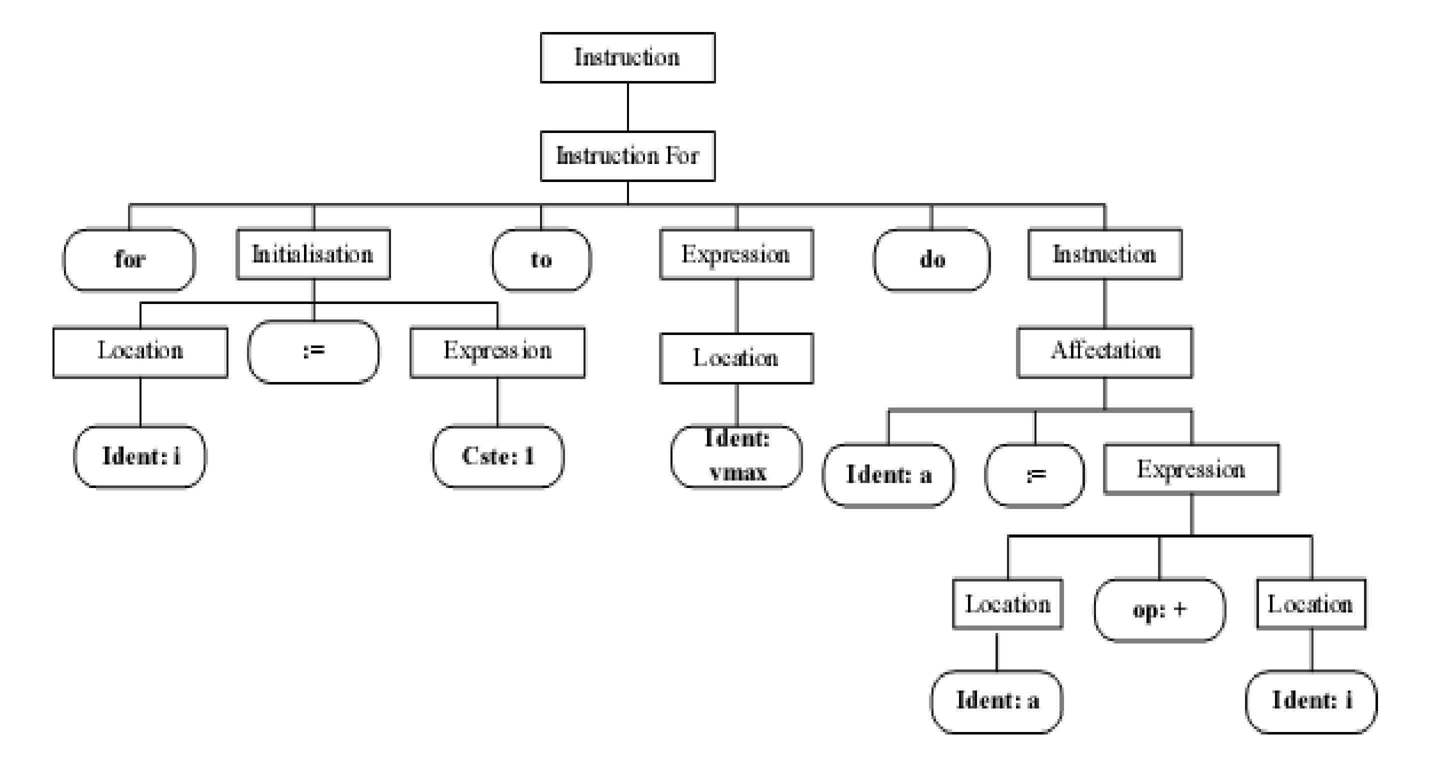
```
for for for i := := 10 do for a a;
```

هنا يأتي دور التحليل النحوي أو l'analyse syntaxique.

2) L’analyse syntaxique:

في هذه المرحلة نتحقق أن ترتيب الـ tokens موافق لتعريف اللغة التي نريد برمجة مترجم لها, نستطيع أن نقول أننا نتحقق من النحو الخاص باللغة و يكون هذا إنطلاقا من مجموعة من القواعد أو ما يسمى grammaire, يقوم المحلل النحوي ببناء شجرة (arbre) باستخدام الـ tokens التي يوفرها المحلل المعجمي:

```
for i := 1 to vmax do a := a+i;
```



- الشجرة النحوية المستخرجة بعد التحليل النحوي -

أما القواعد المحدد للنحو فإنها تكون معرفة على الشكل التالي:

prog -> debut inst fin point

inst ->

| ident affectaion expression

| ...

expression -> ident

| entier

| reel | ...

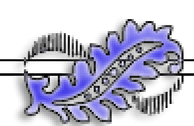
3) L’analyse sémantique:

أثناء التحليل المعنوي (L’analyse sémantique) نتأكد مثلا أن القيمة التي سنسندها لأحد المتغيرات تكون من نفس نوع المتغير, إذ لا يجب إسناد قسمة حقيقية لمتغير صحيح و هكذا.

4) Génération de code:

نقوم هنا بإنتاج كود بلغة الآلة أو لغة التجميع و هذا مثال لكود بلغة قريبة من لغة الآلة:

```
 var_a    A0000                  ; les étiquettes des  variables
 var_i    A0001
 var_vmax  A0002
                                 ; le code du programme
 mov var_i,1
loop :
  mov A0, (var_i)                ; comparaison i >= vmax
  jge A0, (var_vmax), finFor     ; si vrai aller en finFor
  mov A0, (var_a)                ; calcul de a+i
  add A0, A0, (var_i)
  mov var_a,A0                   ; a := a+i
  mov A0, (var_i)                ; on incrémente i
  add A0, A0, 1
  mov var_i, 1
  jmp loop                       ; et on continue la boucle
finFor :
...
```

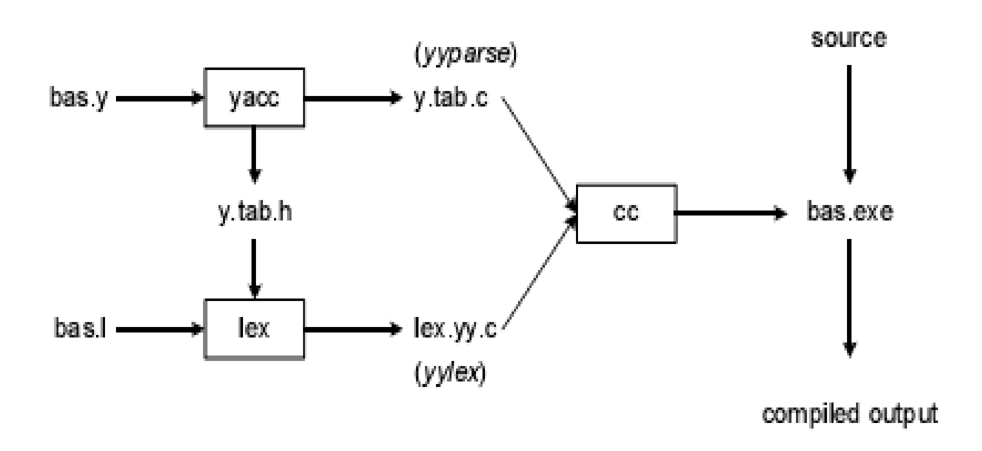


2. LEX و BISON و متطلبات العمل

كانت تلك مقدمة سريعة جدا و مختصرة جدا عن مراحل الترجمة, ماذا بعد, قبل نحدد قواعد اللغة التي سنستخدمها و نبدأ برمجت المحلل المعجمي و النحوي و المعنوي سنلقي نظرة على أداتين مهمتين و هما LEX و BISON.

LEX هي أداة تقوم بتوليد محلل معجمي (Analyseur lexicale) أو ما يسمى بـ Scanner مكتوب باللغة C, يستعمل LEX قوالب (patterns) لمطابقة السلاسل الحرفية الموجودة في الكود المصدر و تحويلها إلى tokens و من ثم إرسالها إلى المحلل النحوي, الـ tokens تكون عبارة عن معرفات عددية ثابتة, أي أن الـ scanner عندما يجد مثلا المتغير x فإنه يرسل للمحلل النحوي الـ token الممثلة للمتغيرات و لتكن IDENT و أيضا يرسل له إسم المتغير و يقوم بإدخاله إلى جدول الرموز table des symboles و تعيين خصائصه كنوعه و غيرها من الخصائص.

BISON أو YACC يولد لنا شفرة باللغة C لمحلل نحوي (Analyseur syntaxique) و يعرف أيضا بالـ parser, يستعمل Bison قواعد اللغة لتحليل الـ tokens القادمة من الـ scanner و يبني عليها شجرة نحوية, تمثل هذه الصورة كيفية التعاون بين الـ LEX و YACC:



إذا و حسب الشكل السابق الملف bas.y يحمل وصف للـ parser أما الملف bas.l فيحمل وصف للـ scanner, الملف y.tab.h يولده الـ YACC و يحوي تعريف الـ tokens وهذا تصريح عن أحد الـ tokens في الملف y.tab.h:

```
#define   IDENT    102
```

بعد ذلك يولد لنا كل من LEX و YACC (سنستعمل BISON و هو مشابه للـ YACC) lex.yy.c و y.tab.c و هما على الترتيب المحلل المعجمي (scanner) و المحلل النحوي (parser).

وباستعمال أحد مترجمات اللغة C نترجم كل من lex.yy.c و y.tab.c لنحصل على البرنامج النهائي bas.exe و هو المترجم الجديد... تهانينا.

ماذا سنحتاج من أدوات لفعل ذلك؟

أولا قم بتحميل TURBO C++ 3.0 من هذا أحد هذه الروابط :

http://www.4shared.com/file/227985692/6671fd89/TC30.html

http://www.mediafire.com/?iyjt4zoej2m

http://www.snapdrive.net/files/618263/CompilerLesson/TC30.zip

قم بتثبيته في القرص \:D بحيث يكون مسار المجلد bin كالآتي: D:\TC\BIN, طبعا أنت حر في تثبيته في أي مكان و لكن الشرح سيكون على أساس أنه مثبت في المسار السابق لأننا سنستعمل البرنامج D:\TC\BIN\TC.EXE لترجمة أكواد C.

LEX و BISON, حملهما من أحد هذه الروابط:

http://www.4shared.com/file/227982793/8d639f91/Lex_Yacc.html

http://www.mediafire.com/?52ym5eyydom

http://www.snapdrive.net/files/618263/CompilerLesson/Lex_Yacc.zip

قم بفك الضغط عن الملف Lex_Yacc.zip وانسخ المجلد Lex_Yacc في القرص \:D, ليصبح لديك المجلد D:\Lex_Yacc\exemples والذي سيكون مسرح الأحداث.

أنسخ محتويات كل من المجلد D:\Lex_Yacc\Bison\bin و المجلد D:\Lex_Yacc\Lex\bin إلى المسار C:\WINDOWS طبعا هذا إذا كانت الويندوز مثبتة في القرص \:C, المهم أن تنسخه إلى المجلد WINDOWS الخاص بالنظام, أخيرا أنسخ المجلد D:\Lex_Yacc\Bison\share بأكمله إلى القرص \:C ليصبح لديك المسار C:\share في جهازك, هنا نكون قد ثبتنا الأدوات اللازمة للعمل, لنبدأ على بركة الله.

قبل البدء في برمجة المترجم سنقوم ببرمجة برنامج صغير باستخدام LEX و BISON و TURBO c++ بحيث يقوم بقراءة ملف يحتوي على عمليات حسابية من عدة سطور, مثلا =5+5 و =3+3*100^20-1002*10 و يقوم بحسابها.

سنعمل على مستوى المجلد D:\Lex_Yacc\exemples, لنبدأ بكتابة المحلل المعجمي أو L’analyseur lexicale أو scanner.

سنقدم لـ LEX وصفا معينا ليولد لنا هو كود C لـ scanner, كيف يكون ذلك الوصف؟ يكون من الشكل التالي:

```
... تعريفات –إذا احتجناها- ...
%%
... قواعد ...
%%
... دوال فرعية –إذا احتجناها- ...
```

الدوال الفرعية عبارة عن كود بلغة C, أما التعريفات و القواعد فلها نحو خاص + بعض أكواد C.

إفتح المفكرة واكتب الكود التالي:

```
%{
#include<stdlib.h>
#include"D:\Lex_Yacc\exemples\expy2.h"
%}
blanc       [ \t]+
chiffre     [0-9]
entier      {chiffre}+
```

قمنا بكتابة التعريفات, لقد احتجنا إلى إضافة كود بلغة C و لذلك كتبناه بين العلامتين **%{** و **%}**, سنحتاج إلى دالة من المكتبة stdlib.h, أما استعمالنا للملف expy.2h فهو لأنه الملف الذي سيحوي تعريفات الـ tokens فيما بعد, لا تقلق فذلك الملف يولده BISON فيما بعد لذلك لا تعره اهتماما الآن, ذلك الجزء الأول من التعريفات, الجزء الثاني -3سطور الأخيرة- قمنا فيه بالتصريح عن بعض القوالب التي سنستعملها لتصفية الـ tokens من الكود المصدر, ماذا تعني؟ لدينا ثلاث قوالب, blanc و chiffre و entier:

blanc	[\t]+	كل الفراغات, فراغ أو أكثر
chiffre	[0-9]	كل الأعداد من 0 إلى 9
entier	{chiffre}+	سلسلة من عدد واحد أو أكثر

هذا جدول للحروف التي نستعملها لإنشاء القوالب و معانيها:

.	كل الأحرف باستثناء \n
\n	سطر جديد
*	صفر نسخة أو أكثر من العبارة السابقة لها

+	نسخة واحدة أو أكثر من العبارة السابقة لها
?	نسخة واحدة أو لا شيء من العبارة السابقة لها
^	بداية السطر
$	نهاية السطر
a|b	a أو b
(ab)+	نسخة واحدة أو أكثر من السلسلة ab
"a+b"	السلسلة a+b حرفيا
[]	فئة من الأحرف, [a-z] تعني كل الأحرف من a إلى z

وكمثال عن بعض الأقنعة لاحظ هذا الجدول:

العبارة	التطابقات
abc	abc
abc*	ab,abc,abcc,abccc,...
abc+	abc,abcc,abccc,...
a(bc)+	abc,abcbc,abcbcbc,...
a(bc)?	a,abc
[abc]	a,b,c
[a-z]	أي حرف بين a و z
[a\-z]	a,-,z
[-az]	-,a,z
[A-Za-z0-9]+	حرف أو أكثر(بما في ذلك الأعداد)
[\t\n]+	الفراغات
[^ab]	أي شيء باستثناء a و b
[a^b]	a,^,b
[a|b]	a,|,b
a|b	a أو b

بهذا يصبح الكود الذي كتبته في المفكرة سابقا أكثر وضوحا و سيتضح كلما تقدمت في قراءة هذا الكتاب, أضف هذا الكود إلى الكود السابق ليصبح:

```
}%
#include<stdlib.h<
#include"D:\Lex_Yacc\exemples\expy2.h"
{%
blanc       [ \t]+
chiffre     [0-9]
entier      {chiffre}+
%%
```

```
{blanc}
{entier}    {
             yylval=atoi(yytext);
             return(NOMBRE);
            }
"+"         return(PLUS);
"*"         return(MULT);
"-"         return(MOIN);
"/"         return(DIVS);
"^"         return(PUIS);
"("         return(PARG);
")"         return(PARD);
"="         return(FIN);
\n          {}
%%
```

أضفنا الجزء الخاص بالقواعد و هو بين العلامتين **%%** و **%%**, هناك 11 قاعدة, ماذا سيفعل الـ LEX ؟ بكل بساطة نلخص القواعد في : **إذا وجدت ... إفعل ...**

مثلا القاعدة الأولى, **إذا وجدت** {blanc} **إفعل** (لا شيء), أي أن الـ scanner سيفوت الفراغات الموجودة في الملف الذي سنقوم بترجمته و حساب ما فيه.

القاعدة الثانية, **إذا وجدت** {entier} **إفعل**:

```
yylval = atoi(yytext);
return(NOMBRE);
```

السطر الثاني ; return(NOMBRE) نعيد فيه الـ token التي حصلنا عليها وهي رقم صحيح حسب القالب المستخدم(entier) إلى المحلل النحوي أو parser, مثلا إذا وجدنا في الملف الذي سنحسب ما بداخله العملية =5+5 فأن أول token نعيدها هي NOMBRE ,ولكن سنحتاج إلى قيمة الـ NOMBRE الذي عثر عليه الـ scanner ولنمررها إلى الـ parser نقوم باسنادها إلى المتغير yylval وهو عبارة عن همزة وصل بين الـ scanner و الـ parser, ولفعل ذلك نحول القيمة الحرفية الموجودة في المتغير yytext الذي يحمل سلسلة الحروف المشكلة لآخر تطابق إلى عدد صحيح باستعمال الدالة atoi الموجودة داخل المكتبة stdlib.h.

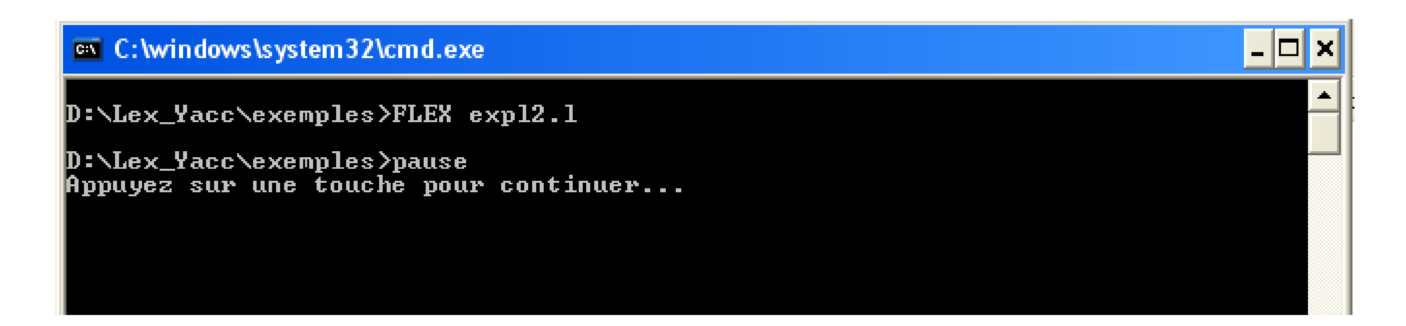
القاعدة الثالثة, **إذا وجدت** + **إفعل** (أعد الـ token التالية: PLUS), قاعدة سهلة وواضحة, إذ سنكتفي بالقول للـ parser أننا وجدنا PLUS ولن نحتاج طبعا إلى قيمتها أو شيء من هذا, فقط نعيد الـ token و نذهب إلى القاعدة التالية.

القاعدة الإخيرة, **إذا وجدت** \n (سطر جديد) **إفعل** (لا شيء).

أنهيانا الآن وصف الـ scanner, ماذا بعد؟ خزن الكود في المسار D:\Lex_Yacc\exemples باسم expl2.l, وباستعمال الأمر FLEX expl2.l يولد LEX كود C للـ scanner, حتى لا نبقى في كل مرة نفتح نافذة الدوس و نكتب الأمر قم بفتح المفكرة و اكتب الأوامر التالية:

```
FLEX expl2.l
Pause
```

ثم أحفظ الملف باسم FLEX_2.bat في المجلد D:\Lex_Yacc\exemples, دوبل كليك على الملف الدفعي flex_2.bat لتحصل على النتيجة التالية:



لا يوجد أخطاء, في حالة وجود أخطاء سيعرض لك LEX رسالة بالأخطاء و مكان كل خطأ و وصفه, لقد قام LEX للتو بتوليد الملف lex.yy.c في المسار D:\Lex_Yacc\exemples والذي يمثل كود C للـ scanner أو L'analyseur lexicale.

ذلك النصف الأول من البرنامج النهائي, لنكمل.

قبل أن نبدأ كتابة وصف YACC سنحدد القواعد (grammaire):

```
Input -> Input Line | £
Line   -> FIN | Exp FIN
Exp    -> NOMBRE
       | Exp  PLUS  Exp
       | Exp  MOIN  Exp
       | Exp  MULT  Exp
       | Exp  DIVS  Exp
       | MOIN Exp
       | Exp  PUIS  Exp
       | PARG Exp   PARD
```

ما هذا ؟؟؟

الرمز £ يعني فراغ أو لاشيء, الكلمات المكتوبة بحروف كبيرة تمثل الـ tokens التي سيعيدها الـscanner , أما باقي الكلمات فهي رموز غير نهائية (symboles non-terminaux).
أحسن طريقة لفهم القواعد السابقة هي تتبع مثال, مثلا هل السلسلة =9*5+5 تحقق شروط الـ grammaire السابق؟

```
Input   -> Input Line
        -> Input
        -> Exp FIN
        -> Exp PLUS Exp FIN
        -> Exp PLUS Exp MULT Exp FIN
        -> NOMBRE PLUS NOMBRE MULT NOMBRE FIN
        -> 5 + 5 * 9 =
```

إذا وصلنا إلى العبارة =9*5+5 إنطلاقا من Input, هذا سيعطيك فكرة مبدئية عن ماهية القواعد التي سنستخدمها لاحقا.

نعود, إفتح المفكرة و اكتب الكود التالي:

```
%{
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "d:\lex_yacc\exemples\expl2.c"
%}
%token NOMBRE PLUS MOIN MULT DIVS PUIS PARG PARD FIN
%left  PLUS   MOIN
%left  MULT   DIVS
%left  NEG
%right PUIS
%start Input
```

بدأنا كتابة الوصف الذي سيولد منه YACC الـ parser, أول ما بدأنا به هو كتابة تعاريف نحتاجها أثناء كتابة القواعد, هناك قسمين من التعاريف, قسم مكتوب باللغة C وهو بين العلامتين }% و {%, واضح الكود المكتوب باللغة C, إذا كنت تتساءل عن الملف expl2.c فهو نفسه الملف lex.yy.c الذي يحمل كود الـ scanner السابق, فقط سنغير اسمه فيما بعد.
القسم الثاني من الكود هو وصف خاص, باستعمال الكلمة المحجوزة %token قمنا بالتصريح عن الـ tokens التي سيستعملها كل من الـscanner و الـ parser, و لدينا 9 tokens:
NOMBRE , PLUS, MOIN, MULT, DIVS, PUIS, PARG, PARD, FIN .
هناك أولوية أثناء إجراء عمليات الحساب, فالقسمة أقوى من الضرب الذي هو أقوى من الجمع و الطرح, وأقصد بكلمة أقوى أولوية الحساب, مثلا عند حساب 6*5+5 فإننا نحسب 6*5 ثم نضيف إلى النتيجة العدد 5, ولهذا أضفنا السطور :

%left PLUS MOIN — أقل أولوية للجمع و الطرح معا, العمليات تجرى على اليسار
%left MULT DIVS — القسمة و الضرب أقوى, نبدأ باليسار أثناء الحساب
%left NEG — النفي أقوى من ما يسبقه
%right PUIS — الرفع إلى قوة أقوى من ما سبقه, ولكن نبدأ باليمين

السطر الأخير %start Input نحدد فيه القاعدة التي نبدأ منها أثناء التحقق من ترتيب الـ tokens القادمة من الـ scanner.

المرحلة الثانية هي كتابة القواعد اللازمة, نغير الكود السابق ليصبح:

```
%{
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "d:\lex_yacc\exemples\expl2.c"
%}
%token NOMBRE PLUS MOIN MULT DIVS PUIS PARG PARD FIN
%left  PLUS   MOIN
%left  MULT   DIVS
%left  NEG
%right PUIS
%start Input
%%
Input:
     |Input Line
     ;

Line :FIN
     |Exp FIN {printf("=%d\n",$1);}
     ;

Exp  :NOMBRE               {$$=$1;}
     |Exp  PLUS  Exp       {$$=$1+$3;}
     |Exp  MOIN  Exp       {$$=$1-$3;}
     |Exp  MULT  Exp       {$$=$1*$3;}
     |Exp  DIVS  Exp       {$$=$1/$3;}
     |MOIN Exp   %prec NEG {$$=-$2;}
     |Exp  PUIS  Exp       {$$=pow($1,$3);}
     |PARG Exp   PARD      {$$=$2;}
%%
```

كل ما فعلناه هو كتابة القواعد و إخبار الـ parser ماذا يفعل عند تحقق كل قاعدة, القاعدة الأولى هي ; | Input : Input Line, وهي نقطة البداية, Input ستعطي شيئين, إما فراغ أو Input Line, سهلة وواضحة.

القاعدة الثانية, Line, إما تعطينا FIN (وهي الحرف "=" كما هو معرف على مستوى وصف LEX) أو Exp FIN, وهنا نكتب نتيجة العملية الحسابية باستعمال ; printf("=%d\n",$1), ماذا يعني الرمز $1 ؟

أثناء التحقق يمكننا أن نعطي و نأخذ قيم الرموز و الـ tokens التي نجدها في طريقنا, وهنا سنكتب على الشاشة قيمة Exp وهي معرفة بـ $1, أما قيمة FIN إذا أردنا إستعمالها فهي $2 وهكذا.

في كل قاعدة هناك نصف أيمن و نصف أيسر, و لإسناد قيم أو قراءة قيم الرموز المكونة للقاعدة فإننا نستعمل $x:

Exp -> Exp PLUS Exp
$$ $1 $2 $3

طبعا استعمال قيم الرموز باستخدام $x أو $$ هو تابع للتحليل المعنوي(L'analyse sémantique) فالمحلل النحوي أو L'analyseur syntaxique لا يهتم بالقيم.

ننتقل إلى القاعدة الثالثة الخاصة ب Exp, Exp تعطينا أحد ثمانية خيارات:

- في حالة Exp -> NOMBRE فالأمر بسيط, نعطي قيمة الرقم إلى Exp باستعمال ; $$=$1, و تسمى هذه الحركة الأخيرةبـ Action sémantique كما أذكر.

- في حالة Exp -> Exp PLUS Exp فالأمر بسيك أيضا, Exp التي عى يسار القاعدة تأخذ قيمة مجموع Exp التي على يمين القاعدة و نفعل هذا باستعمال $$=$1+$2 , وهكذا نفس الشيء بالنسبة لباقي القواعد المماثلة

- في حالة Exp -> MOINS Exp هناك شيء جديد, طبعا نقصد من هذه القاعدة حالة النفي مثلا -112, ولكن تلك الناقص ليست هي نفسها عملية الطرح فهي لها أولوية قصوى, مثلا إذا وجدنا -5/6 فإننا نحسب -5 أولا, ولهذا نعطيها أولوية NEG باستعمال التعليمة prec NEG%, أما الباقي واضح وهو إعطاء القيمة –Exp إلى Exp التي على اليسار باستعمال; $$ = $2 .

- بالنسبة للحالة Exp -> Exp PUIS Exp فنقصد بها الرفع إلى القوة(3^2=9) و سنحسبها باستعمال الدالة pow الموجودة في المكتبة math.h كما يلي ; $$=pow($1, $3).

إلى هنا ننهي قسم القواعد الخاصة بالـ parser, بقي لنا أن نصرح عن الدالة الرئيسية ()main وهذا في قسم الدوال الفرعية:

```
%{
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "d:\lex_yacc\exemples\expl2.c"
%}
%token NOMBRE PLUS MOIN MULT DIVS PUIS PARG PARD FIN
%left  PLUS   MOIN
%left  MULT   DIVS
%left  NEG
%right PUIS
%start Input
%%
Input:
     |Input Line
     ;

Line :FIN
     |Exp FIN {printf("=%d\n",$1);}
     ;

Exp  :NOMBRE          {$$=$1;}
     |Exp  PLUS  Exp      {$$=$1+$3;}
     |Exp  MOIN  Exp      {$$=$1-$3;}
     |Exp  MULT  Exp      {$$=$1*$3;}
     |Exp  DIVS  Exp      {$$=$1/$3;}
     |MOIN Exp   %prec NEG {$$=-$2;}
     |Exp  PUIS  Exp      {$$=pow($1,$3);}
     |PARG Exp   PARD     {$$=$2;}
%%
int yyerror (char *s)
{
printf("%s\n",s);
}
int yywrap(){
return 1;
}
main()
{
clrscr();
if((yyin=fopen("d:\\lex_yacc\\exemples\\input.txt","r"))==NULL)
```

```
{
 printf("input.txt not found !\n");
 getch();
 return;
}
yyparse();
getchar();
}
```

قمنا بتعريف أجسام الدالتين ()yyerror التي تستدعى من طرف الـ parser عند وقوع خطأ أو عدم تطابق ترتيب الـ tokens المرسلة من طرف الـ scanner مع القواعد المحددة, الدالة الثانية هي ()yywrap وتستدعى عند نفاذ المدخلات وفي حالتنا هذه عند نفاذ العمليات المراد حسابها.

الدالة ()main, ماذا بها؟

أولا نقوم بفتح الملف input.txt -الذي سيحتوي على العمليات المراد حسابها- باستعمال ()fopen التي تعيد إلينا مؤشر للملف, نسند ذلك المؤشر إلى المتغير yyin و هو متغير معرف مسبقا من طرف YACC و يمثل ملف المدخلات, و في حالت وجود خطأ أثناء فتح الملف input.txt نعرض رسالة خطأ و نتوقف.

بعد ذلك نستدعي الدالة ()yyparse والتي تمثل هنا الـparser وتقوم بكل العمل الذي وصفناه سابقا.

الآن أحفظ ما كتبناه من وصف في المفكرة إلى الملف D:\Lex_Yacc\exemples**expy2.y**, وبنفس الطريقة التي استعملنا بها LEX سنستعمل BISON, عوضا عن كتابة أوامر في نافذة الدوس أنشئ ملف دفعي جديد باسم BISON_2.bat داخل المجلد \D:\Lex_Yacc\exemples واكتب به ما يلي:

```
BISON -d expy2.y
Pause
```

دوبل كليك على الملف الدفعي الجديد و ستحصل على هذه النتيجة:

```
C:\windows\system32\cmd.exe

D:\Lex_Yacc\exemples>BISON -d expy2.y
expy2.y:31: warning: previous rule lacks an ending `;'

D:\Lex_Yacc\exemples>pause
Appuyez sur une touche pour continuer... _
```

أيضا سيولد BISON ملفين و هما expy2.tab.c و expy2.tab.h, الملف expy2.tab.c هو الملف الذي به الدالة ()main وبالتالي هو الملف الذي سنترجمه باستخدام ++TURBO C, أما الملف expy2.tab.h فهو يحتوي على تعريفات الـ tokens من أجل استعماله داخل

الـscanner و إذا كنت تذكر ففي ملف وصف السكانر expl2.l هناك هذا السطر لاستعمال ملف تعريفات الـ tokens:

```
%{
#include<stdlib.h>
#include"D:\Lex_Yacc\exemples\expy2.h"
%}
```

ولكن اسمه expy2.tab.h و ليس expy2.h !! ليست مشكلة, سنغير اسمه إلى expy2.h وانتهى الأمر.

نفس الشيء بالنسبة لملف وصف البارسر expy2.y لدينا:

```
%{
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "d:\lex_yacc\exemples\expl2.c"
%}
```

أيضا ملف السكانر الناتج عن LEX اسمه lex.yy.c وليس expl2.c !!! نفس الشيء, سنغير الأسماء فقط و لفعل هذا أضف ملف دفعي جديد باسم rename.bat وليكن محتواه :

```
ren lex.yy.c expl2.c
ren expy2.tab.c expy2.c
ren expy2.tab.h expy2.h
```

دوبل كليك على الملف الدفعي rename.bat لتتغير أسماء الملفات lex.yy.c و expy2.tab.c و expy2.tab.h إلى expl2.c و expy2.c و expy2.h على الترتيب.

إذا و في كل مرة نغير من وصف الملف expl2.l أو expy2.y فإننا نحذف ملفات الكود القديمة (expl2.c و expy2.c و expy2.h) و نعيد تنفيذ كل من FLEX_2.bat و BISON_2.bat و rename.bat.

بعد حصولنا على كود سورس السكانر و البارسر نترجمهما باستعمال TURBO C 4.5 وتحديد باستعمال البرنامج D:\TC\BIN\TC.EXE وكي نتجنب فتح TC في كل مرة فإننا نفعل ذلك من سطر الأوامر, أنشئ ملف دفعي جديد بإسم compile.bat واكتب فيه هذا الأمر الذي يترجم الكود:

```
D:\TC\BIN\TC.EXE D:\Lex_Yacc\exemples\expy2.c /b
pause
```

نفذ الملف compile.bat و ستتم ترجمة الكود و ستحصل على ملف تنفيذي باسم EXPY2.EXE وهو البرنامج النهائي و لكي نرى نتيجة عملنا أضف ملف جديد –طبعا نحن لا زلنا

و سنبقى نعمل على مستوى المجلد D:\Lex_Yacc\exemples- باسم input.txt ليحمل المدخلات و لنضف اليه بعض العمليات التي نريد حسابها, مثلا:

```
5+5=
10*55=
3+2^2=
-5+(11*8)-16=
```

نفذ الآن برنامجنا الجديد EXPY2.EXE و ستحصل على هذه النتيجة:

```
D:\Lex_Yacc\exemples\EXPY2.EXE
=10
=550
=7
=67
```

لقد قام بحساب كل العمليات الموجودة في الملف input.txt, لا تعتقد أن هذا شيء لا أهمية له فهذه أول خطوة لبرمجة المترجم الخاص بنا, يجب أن تكون الرؤيا قد اتضحت الآن عن كيف سنعمل منذ الآن فصاعدا, لنكتب عملية خاطئة -أو بالأحرى لا تتوافق مع القواعد المحددة- لنرى النتيجة, غير الملف input.txt ليصبح:

```
5+5=
10*55=
3+2^2=
-5++(11*8)-16=
```

أكيد لقد لاحظت أين الخطأ, بعد تنفيذ EXPY2.EXE سنحصل على هذه النتيجة:

```
D:\Lex_Yacc\exemples\EXPY2.EXE
=10
=550
=7
parse error
```

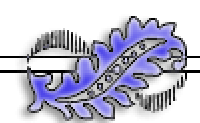
و تظهر نتيجة الخطأ في السطر الرابع, لاحقا سنرى كيف نعرض رسالة بالخطأ المحدد و رقم السطر الذي وقع فيه الخطأ إن شاء الله.

لتحميل المجلد exemples الذي يحتوي على هذا المثال استخدم أحد هذه الروابط:

http://www.4shared.com/file/227983170/36d8f872/exemples1.html

http://www.mediafire.com/?gqniqohtmyz

http://www.snapdrive.net/files/618263/CompilerLesson/exemples1.zip



3. برمجة المترجم

من هنا نبدأ برمجة المترجم, أول ما يجب تحديده هو اللغة التي سنستخدمها, سنستخدم نفس النحو المستعمل لكتابة الـ Algorithmes, وكمثال عن لغتنا:

```
algorithme alg
entier resultat,a;
debut
ecrire "Entrer a = ";
lire a;
resultat<-5;
a<-resultat*10 ;
ecrire !,"a = ",a;
lire a;
fin.
```

الرمز (!) نتفق أنه يعني سطر جديد, أي أكتب الجملة الموالية في سطر جديد, مبدئيا لغتنا الجديدة لا تحتوي على حلقات, فقط إسناد و كتابة و قراءة.

سنسمي هذا المترجم compalg اختصارا لـ **Com**pilateur d'**alg**orithmes, هذه اللغة تحتاج إلى grammaire أو مجموعة القواعد التي تحددها و لنبدأ بهذا الـ grammaire و سنغيره كلما أردنا إضافة قاعدة جديدة:

```
prog -> ALGO IDENT declaration debut
declaration -> £ | decl_type  ident  POINT_VER declaration
decl_type -> DEC_ENTIER | DEC_REEL | DEC_CARA | DEC_CHAINE
ident -> IDENT | IDENT VER ident
debut -> DEBUT command_seq
command_seq -> £
          | command_seq  FIN   POINT
          | affect   POINT_VER command_seq
          | READ  read          POINT_VER command_seq
          | WRITE write         POINT_VER command_seq
Affect -> IDENT AFFECT fexpr | IDENT AFFECT sexpr
read -> IDENT | IDENT VER read
write -> writed_expr | writed_expr VER write
writed_expr -> NEW_LINE | fexpr | sexpr
fexpr -> REEL
    |ENTIER
    |IDENT
    |fexpr PLUS fexpr
    |fexpr MOIN fexpr
    |fexpr MULT fexpr
```

```
    |fexpr DIVS fexpr
    |MOIN fexpr
    |fexpr PUIS fexpr
    |PARG fexpr PARD
sexpr -> CHAINE | CARA
```

تذكر أن الرمز (£) يعني اللاشيء, الرموز بالحروف الكبيرة هي الـ tokens التي يرسلها البارسر أو L'analyseur lexicale لذلك هي رموز نهائية(symboles terminaux).
عد إلى المجلد D:\Lex_Yacc\exemples واحذف منه الملفات التي استخدمناها في المثال السابق و لنبدأ على بركة الله, افتح المفكرة و نبدأ كتابة الكود الذي يمثل وصف السكانر:

```
%{
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include"d:\lex_yacc\exemples\compalg.h"
int line=1;
%}
```

سنحتاج كل من تلك المكاتب, الملف compalg.h هو الذي سيولده BISON فيما بعد و الذي يحوي التصريح عن الـ tokens اللازمة, المتغير line يمثل السطر الحالي الذي نحن بصدد تحليله و استخراج الـ tokens منه و سنزيد من قيمتخ عند نهاية كل سطر و هذا لنعرف موقعنا إذا صادفنا خطأ ما, واضح.
لنعرف الآن القوالب اللازمة لاستخراج الـ tokens من الكودسورس:

```
blanc          [ \t]+
nbr            [0-9]+
entier         {nbr}
reel           {entier}\.{nbr}
ident          [a-zA-Z_]([0-9a-zA-Z_])*
a [aA]
b [bB]
c [cC]
d [dD]
e [eE]
f [fF]
g [gG]
h [hH]
i [iI]
j [jJ]
k [kK]
```

```
l [lL]
m [mM]
n [nN]
o [oO]
p [pP]
q [qQ]
r [rR]
s [sS]
t [tT]
u [uU]
v [vV]
w [wW]
x [xX]
y [yY]
z [zZ]
```

أعتقد أن القوالب السابقة تشرح نفسها, مثلا nbr يمثل كل الأعداد من 1 إلى 9 مرة واحدة على الأقل أو أكثر, أي أنه مثلا 14522 تنتمي إلى ذلك القالب.
القالب entier هو نفسه nbr, أما القالب reel فيختلف قليلا, إذ أنه عبارة عن nbr.nbr مثلا 12.11 و غيرها من الأعداد الحقيقية التي تندرج تحت هذا القالب.
القالب ident الذي يمثل المتغيرات –مثل x أو y_10- فتكون بدايته عبارة عن أحد الحروف التي تنتمي إلى المجال[a-z] أو [A-Z] أو الحرف (-) و هذا حسب [a-zA-Z_] , ثم تليه مجموعة أخرى من الحروف التي تكون ضمن [a-z] أو [A-Z] أو [0-9] أو (_), النصف الثاني من المتغير يمكن أن يكون أو لا يكون و حددنا هذا بـ *([0-9a-zA-Z_]).
أما بقية القوالب من a إلى z فقد استعملناها فقط حتى تكون الحروف الكبيرة في لغتنا مثلها مثل الحروف الصغيرة, تماما كلغة باسكال التي لا تفرق بين الحروف الكبيرة و الصغيرة, مثلا القالب a يمكن أن يكون a أو A.

لننتقل إلى الجزء الثاني و هو تعريف القواعد التي من الشكل إذا وجدت... إفعل...

```
%%
"\"".""\""                    return(CARA);
"\""(.)+"\""                  return(CHAINE);
{a}{l}{g}{o}{r}{i}{t}{h}{m}{e} return(ALGO);
{d}{e}{b}{u}{t}               return(DEBUT);
{f}{i}{n}                     return(FIN);
{e}{n}{t}{i}{e}{r}            return(DEC_ENTIER);
{r}{e}{e}{l}                  return(DEC_REEL);
{c}{a}{r}{a}{c}{t}{e}{r}{e}   return(DEC_CARA);
{c}{h}{a}{i}{n}{e}            return(DEC_CHAINE);
{l}{i}{r}{e}                  return(READ);
{e}{c}{r}{i}{r}{e}            return(WRITE);
```

```
"!"                              return(NEW_LINE);
\.                               return(POINT);
;                                return(POINT_VER);
,                                return(VER);
\n                               {line++;}
"<-"                            return(AFFECT);
"("                             return(PARG);
")"                             return(PARD);
"+"                             return(PLUS);
"-"                             return(MOIN);
"*"                             return(MULT);
"/"                             return(DIVS);
"^"                             return(PUIS);
{ident}                          return(IDENT);
{reel}                          return(REEL);
{entier}                        return(ENTIER);
{blanc}
%%
```

لن أشرحها لبساطتها, الآن أنهينا وصف السكانر, خزن الملف الجديد باسم lcompalg.l طبعا و دائما داخل المجلد D:\Lex_Yacc\exemples, و أيضا أضف ملفا دفعيا جديدا باسم FLEX_2.bat و اكتب به الأمر :

```
FLEX lcompalg.l
```

دوبل كليك على FLEX_2.bat تحصل على الملف lex.yy.c كنتيجة.

لننتقل إلى بناء النصف الثاني من البرنامج وهو البارسر أو L'analyseur syntaxique, افتح المفكرة و أكتب :

```
%{
#include<conio.h>
#include<math.h>
#include "d:\lex_yacc\exemples\lcompalg.c"
int errors=0;
%}
```

كالمعتاد, سنحتاج إلى دوال من تلك المكاتب أما بالنسبة للملف lcompalg.c فهو نفسه الملف lex.yy.c –سنغير تسميته لاحقا-, سنستعمل المتغير errors لحساب عدد الأخطاء الموجودة في الملف الذي يحوي الـ Algorithm, بعد ذلك نضيف مزيدا من التعريفات:

```
%token    IDENT
```

```
%token    ENTIER
%token    REEL
%token    CHAINE
%token    CARA
%token ALGO DEBUT FIN POINT POINT_VER VER
%token DEC_ENTIER DEC_REEL DEC_CARA DEC_CHAINE
%token AFFECT READ WRITE REEL PARG PARD
%token PLUS MOIN MULT DIVS PUIS NEW_LINE

%left  PLUS MOIN
%left  MULT DIVS
%right PUIS
%left  NEG
%start prog
```

في التسع سطور الأولى قمنا بالتصريح عن الـ tokens اللازمة, في الأربع سطور التالية عرفنا أولوية كل عملية حسابية, السطر الأخير فيه القاعدة التي تمثل نقطة الانطلاق, يا للوضوح...

الخطوة التالية هي التصريح عن القواعد المتبعة في هذه اللغة:

```
%%
prog:ALGO IDENT declaration debut
    ;
declaration:
           |decl_type  ident  POINT_VER declaration
           ;
decl_type:DEC_ENTIER
         |DEC_REEL
         |DEC_CARA
         |DEC_CHAINE
         ;
ident:IDENT
     |IDENT VER ident
     ;
debut:DEBUT command_seq
     ;
command_seq:
           |command_seq  FIN    POINT
           |affect   POINT_VER command_seq
           |READ  read          POINT_VER command_seq
           |WRITE write         POINT_VER command_seq
           ;
affect: IDENT AFFECT fexpr
```

```
      | IDENT AFFECT sexpr
      ;
read:IDENT
    |IDENT VER read
    ;
write:writed_expr
     |writed_expr VER write
     ;
writed_expr:NEW_LINE
           |fexpr
           |sexpr
           ;
fexpr:REEL
    |ENTIER
    |IDENT
    |fexpr PLUS fexpr
    |fexpr MOIN fexpr
    |fexpr MULT fexpr
    |fexpr DIVS fexpr
    |MOIN fexpr %prec NEG
    |fexpr PUIS fexpr
    |PARG fexpr PARD
    ;
sexpr:CHAINE
    |CARA
    ;
%%
```

نحن نحقق تقدما سريعا هنا, الكود السابق هو نفسه الـ grammaire الذي اتفقنا على استعماله سابقا إلا أن شكله تغير قليلا, لا تلمني لكن ألق اللوم على BISON لأنه هو من يريد هذا الشكل للـ grammaire .

لم نستعمل Les actions sémantiques وهذا لأننا حاليا نحن بصدد إنجاز السكانر و البارسر فقط أو L'analyseur lexicale و L'analyseur syntaxique فقط, لكل أوانه.

القطعة الباقية من الكود اللازم هي التصريح عن الدالة الرئيسية و غيرها في الشطر الثالث من ملف وصف البارسر, أضف هذا الكود إلى المفكرة:

```
int yyerror (char *s)
{
   errors++;
```

```
   printf("Erreur :syntaxe erreur: ligne %d\n",line);
}
int yywrap(){return 1;}
main(int argc,char *argv[])
{
clrscr();
if((yyin=fopen("d:\\lex_yacc\\exemples\\test.alg","r"))==NULL)
{
 printf("test.alg not found !\n");
 getch();
 return;
}
yyparse();
if(!errors) printf("Ok");
getch();
return;
}
```

ما الجديد عن المثال السابق؟
عند وقوع خطأ ما يقوم البارسر باستدعاء الدالة yyerror() وهنا نعرض نحن رسالة الخطأ, السطر الذي وقع فيه الخطأ هو ذو الرقم line وهذا المتغير معرف في الملف lcompalg.c و قد صرحنا عليه في lcompalg.l, و عند وقوع كل خطأ نزيد من قيمة المتغير errors.
الدالة yywrap() تبقى كما هي, أما الدالة main() فنقوم فيها بفتح الملف test.alg الذي يحوي الـ Algorithme الذي نريد ترجمته, وبعد تحليل الملف سنعرض الرسالة Ok إذا كانت قيمة المتغير errors مساوية للصفر.

الآن خزن الكود السابق في ملف باسم compalg.y و أضف ملفا دفعيا جديدا باسم BISON_2.bat و أكتب فيه هذا الأمر:

```
BISON -d compalg.y
```

نفذ الملف BISON_2.bat لتحصل على ملفين اثنين, compalg.tab.c و compalg.tab.h, قبل أن نترجم الكود المولد من طرف LEX و BISON سنغير أسماء الملفات lex.yy.c و compalg.tab.c و compalg.tab.h إلى lcompalg.c و compalg.c و compalg.h على الترتيب – لا تسأل لماذا, فقط تروق لي الأسماء الجديدة – وهذا باستعمال هذه الأوامر:

```
ren lex.yy.c lcompalg.c
ren compalg.tab.c compalg.c
ren compalg.tab.h compalg.h
```

الأوامر السابقة هي أوامر DOS وعندما نريد تنفيذها لن نفتح موجه أوامر دوس و ننفذها فقط نضيف ملف دفعي باسم rename.bat و نكتب فيه الأوامر السابقة, دوبل كليك عليه و انتهى الأمر.

نفس الشيء بالنسبة لعملية ترجمة الكود النهائي, أضف ملف دفعي باسم compile.bat و اكتب فيه هذا الأمر:

```
D:\TC\BIN\TC.EXE D:\Lex_Yacc\exemples\compalg.c /b
pause
```

لقد قمنا بترجمة كود C باستعمال سطر الأوامر فأنا أكره فتح TURBO C++ و ترجمة الكود بتلك الطريقة, دوبل كليك على compile.bat وانتهى الأمر.

إذن, بعد تنفيذ الملف compile.bat سنحصل على البرنامج النهائي وهو COMPALG.EXE, و أثناء كتابة وصف البارسر قمنا باستعمال الملف test.alg كمدخلات للمترجم عن طريق هذا السطر:

```
//…
if((yyin=fopen("d:\\lex_yacc\\exemples\\test.alg","r"))==NULL)
//…
```

ذلك و قبل تنفيذ المترجم, أنشئ ملف باسم test.alg و اكتب فيه نص الـ Algorithme الذي نحن بصدد ترجمته, وهذا هو الكود:

```
algorithme alg
entier resultat,a;
debut
ecrire "Entrer a = ";
lire a;
resultat<-5;
a<-resultat*10 ;
ecrire !,"a = ",a;
lire a;
fin.
```

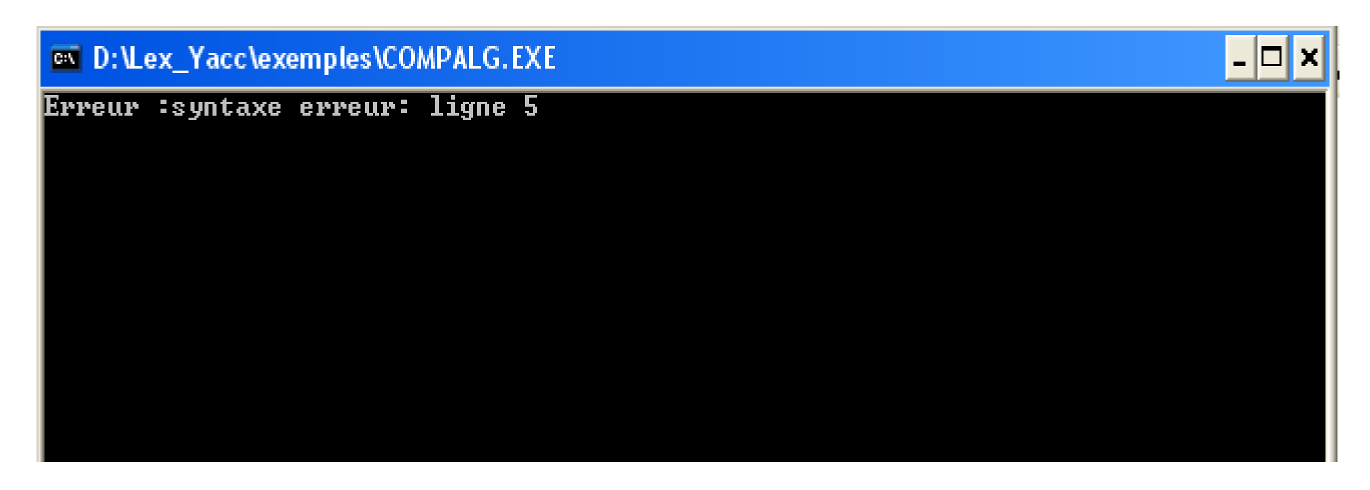
بعد ذلك نفذ برنامج المترجم COMPALG.EXE حتى يتفحص المترجم, هذه هي نتيجة التنفيذ:

```
D:\Lex_Yacc\exemples\COMPALG.EXE
Ok
```

لقد قام المترجم بتفحص الكود و وجد أنه مطابق للقواعد المحدد أثناء وصف البارسر و لذلك عرض رسالة (Ok), لنجرب كتابة Algorithme خاطئ, مثلا غير السطر الخامس ليصبح :

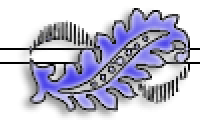
```
algorithme alg
entier resultat,a;
debut
ecrire "Entrer a = ";
lire a,;
resultat<-5;
a<-resultat*10 ;
ecrire !,"a = ",a;
lire a;
fin.
```

هناك فاصلة إضافية, و بما أن هذا لا ينطبق مع قواعد اللغة التي حددناها سابقا ستكون النتيجة:



لقد اكتشف الخطأ !!! والسطر الذي وقع فيه الخطأ !!!... عجيب (وكأننا نعتمد على الحظ هنا).

من المؤكد أنه قد تكونت لديك فكرة واضحة جدا عن مبدأ عمل L'analyseur lexicale و L'analyseur syntaxique (Scanner & Parser).



ننتقل الآن إلى المرحلة التالية, ألا وهي برمجة L'analyseur sémantique أو عملية التحليل المعنوي, نغير الـ Algorithme المراد ترجمته - محتوى الملف test.alg- إلى :

```
algorithme alg
entier resultat,a;
reel r;
caractere c;
chaine s, r;
debut
ecrire "Entrer a = ";
lire a;
lire r;
c<-"a";
s<-"chaine s";
c<-s;
c<-"hhhh";
resultat<-5;
a<-resultat*10 ;
ecrire !,"a = ",a;
a<-r;
a<-0.5;
x<-150;
lire a;
fin.
```

أضفنا بعض التصريحات و التعليمات, طبعا لا معنى لها فنحن فقط نجر و لسنا نكتب في برنامج ذو هدف, أضفنا المتغير الحقيقي r والمتغير c من نوع حرف, أما المتغير s فهو سلسلة حرفية, ما رأيك هل يوجد أخطاء في باقي الكود؟

أجل هناك أخطاء معنوية (Des erreurs sémantiques), مثلا في السطر5 المتغير r معرف مسبقا عل أنه reel و في السطر 12 و 13 لا يمكننا إسناد سلسلة حرفية إلى متغير من نوع caractere, أيضا في السطر 16 و 17 لا يمكننا وضع قسمة حقيقية في متغير صحيح, وفي السطر 18 إذ قمنا بإسناد قيمة للمتغير x مع أننا لم نصرح عنه من قبل, لكن, إلى حد الآن المترجم لا يستطيع الكشف عن هذه الأخطاء, خزن الملف test.alg و شغل البرنامج COMPALG.EXE و سيقول لك: لا يوجد أخطاء, هنا يأتي دور L'analyseur sémantique.

هنا يجب علينا بناء la table des symboles أو جدول الرموز, سنخزن فيها كل المتغيرات و أنواعها, قيمها لن تهمنا, أضف ملف جديد باسم SYMB_TAB.H داخل المجلد D:\Lex_Yacc\exemples و افتحه و لنبدأ بكتابة الكود اللازم.

هل تذكر المتغير yylval من المثال الأول لكيفية استعمال LEX و YACC ؟ لقد قلنا عنه أنه حلقة وصل بين LEX و YACC إذ أن LEX إذا وجد مثلا رقما صحيحا فإننا نستطيع تمرير قيمة ذلك المتغير للـ YACC مع الـ token, نحن هنا سنحتاج إلى تمرير السلسلة الحرفية التي تمثل المتغير حتى نتمكن من تخزينها في قائمة المتغيرات و التعامل معها, حاليا LEX و حسب الوصف الذي كتبناه و تحديدا في هذا السطر :

```
{ident}          return(IDENT);
```

لا يمرر للـ YACC إلا الـ token التي تمثل المتغير و هي IDENT, سنمرر المتغير نفسه عن طريق المتغير المحجوز yylval و لكن نوعه int وليس char[] ! لا يهم, بامكاننا التحكم في نوعه.

لن نغير نوع yylval من int إلى char أو غيره فنحن سنمرر للـ YACC عدة أنواع, الحل هو في استخدام union, أكيد أنت تعرفها, و إذا لم تكن تعرفها فهي شبيهة بالـ struct ولكن الفرق يكمن في أن مكونات الـ union تشغل نفس المساحة من الذاكرة, مثلا لاحظ هذه الـ union:

```
typedef union{
 char  Tstr[128];
 int   Tint;
 float Tfloat;
} new_type;
new_type t;
```

إذا قمنا باسناد قيمة لـ t.Tstr ثم بعد ذلك أسندنا قيمة لـ t.Tint فإن قيمة t.Tstr ستضيع لأن المتغيرات الثلاث يتشاركون في الذاكرة المحجوزة لهم وهي الذاكرة اللازمة لتخزين أكبر متغير و هنا هي مساوية لـ 128بايت.

كيف نجعل yylval ينتمي إلى ذلك النوع؟ افتح الملف compalg.y و أضف إليه هذا المقطع في المكان المبين:

```
%{
#include<conio.h>
#include<math.h>
#include "d:\lex_yacc\exemples\lcompalg.c"
%}
%union{
 char  Tstr[128];
 int    Tint;
 float  Tfloat;
}
%token    IDENT
//…
```

طبعا الجزء المضلل هو الجديد في الكود, أغلق الملف و أحفظه طبعا.

وبهذا نكون قد غيرنا نوع yylval إلى النوع الذي نحتاجه, بقي لنا أن نمرر اسم المتغير إلى YACC كلما وجدناه وهذا على مستوى الملف lcompalg.l:

```
//...
"^"             return(PUIS);
{ident}         {
                    strcpy(yylval.Tstr,yytext);
                    return(IDENT);
                }
{reel}          return(REEL);
//...
```

إذن و قبل أن نعيد IDENT على شكل token إلى YACC قمنا بتمرير قيمة المتغير إلى YACC و هذا بنسخها في yylval.Tstr, طبعا قلنا سابقا أن قيمة كل تطابق نجدها في المتغير المحجوز yytext, أحفظ و أغلق lcompalg.l ولننتقل إلى الخطوة الموالية.

سنبدأ بمعالجة الخطأ المعنوي الأول و هو تكرار التصريح عن المتغيرات كما حدث مع المتغير r في المثال السابق, افتح الملف SYMB_TAB.H و لنبدأ بإضافة هذا الكود:

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>

typedef struct sym_node_
{
 char name[56];
 struct sym_node *next;
}sym_node;

sym_node *sym_table=NULL;
```

عرفنا التركيبة sym_node المتكونة من اسم المتغير و مؤشر لتركيبة من نفس النوع, و هنا أردنا إنشاء قائمة ديناميكية من المتغيرات, حاليا يهمنا فقط اسم المتغير, بعد ذلك صرحنا عن sym_table وهي القائمة التي تمثل جدول المتغيرات هنا و هي مؤشر لتركيبة من نوع sym_node, طبعا هذا من أساسيات البرمجة بـلغة C ولو كنت تجهل ماذا يعني ذلك الكود لما كنت تقرأ في هذا الكتاب.

لنكمل, سنضيف دالتين, الأولى put_sym والتي تثبت متغير جديد في القائمة, أما الثانية get_sym ومن اسمها نستنتج أنها ستعيد إما مؤشر للتركيبة التي تحوي المتغير المحدد على شكل بارامتر أو ستعيد NULL إذا كان المتغير غير مثبت من قبل:

```
sym_node *put_sym(char *sym_name)
{
 sym_node *ptr;
 ptr=(sym_node*)malloc(sizeof(sym_node));
 strcpy(ptr->name,sym_name);
 ptr->next=(sym_node*)sym_table;
 sym_table=ptr;
 return ptr;
}

sym_node *get_sym(char *sym_name)
{
 sym_node *ptr;
 for(ptr=sym_table;ptr!=NULL;ptr=(sym_node*)ptr->next)
         if(!strcmp(ptr->name,sym_name))return ptr;
 return NULL;
}
```

كود واضح, كيف نستعمله؟ أغلق و أحفظ تغيرات الملف SYMB_TAB.H ولنعد إلى ملف وصف البارسر compalg.y, افتحه ولنبدأ بإضافة بعض الأكواد:

```
%{
#include<conio.h>
#include<math.h>
#include "d:\lex_yacc\exemples\lcompalg.c"
#include "d:\lex_yacc\exemples\SYMB_TAB.H"
int errors=0;
void setup_sym(char* sym_name)
{
 sym_node *sym;
 sym=get_sym(sym_name);
 if(sym==NULL) put_sym(sym_name);
 else
 {
  errors++;
  printf("Erreur: %s est deja definie : ligne
%d.\n",sym_name,line);}
}

int sym_check(char* sym_name)
{
 if(get_sym(sym_name)==NULL)
  {
   errors++;
```

```
    printf("Erreur: %s est inconnu : ligne %d.\n",sym_name,line);
    return 0;
   }
  return 1;
}
%}
//…
```

بسيطة, الدالة setup_sym تقوم بتثبيت متغير داخل جدول المتغيرات و هذا باستعمال الدوال المعرفة داخل الملف d:\lex_yacc\exemples\SYMB_TAB.H, قبل أن نضيف متغيرا جديدا يجب أن نتأكد من أنه لم يضاف من قبل بواسطة الدالة get_sym, إذا أعادة الدالة get_sym قيمة غير NULL فإننا نعرض رسالة خطأ تفيد أن المتغير معرف من قبل و في أي سطر وقع الخطأ, المتغير line عرفناه من قبل داخل الملف lcompalg.l الذي يحمل وصف السكانر و بالتالي هو معرف في الملف lcompalg.c المولد من طرف LEX, أما إذا أعادت الدالة get_sym القيمة NULL فإننا نثبت المتغير الجديد والسلام.

أما الدالة sym_check فنستدعيها لنتأكد أن المتغير المعطى لها على شكل بارامتر مثبت مسبقا في قائمة المتغيرات, و هذا لتجنب استعمال متغيرات غير مصرح بها من قبل, إذا لم يكن المتغير مثبتا من قبل فإننا نعرض رسالة خطأ.

الآن سنرى أين نستدعي الدوال, نستدعيها كلما وجدنا متغيرا أثناء تحليل الكود, لاحظ هذه الإضافات على الملف compalg.y:

```
//...
%token <Tstr>   IDENT
//...
```

حددنا نوع القيمة التي ستأتي مع الـ token الخاص بالمتغيرات IDENT, وهي Tstr, أي أنه $x الموافقة للـ token عبارة عن سلسلة حرفية, لنواصل:

```
//...
declaration:
           |decl_type  ident  POINT_VER declaration
           ;
//...
ident:IDENT              {setup_sym($1);}
     |IDENT VER ident    {setup_sym($1);}
     ;
//...
```

كما ترون فإن القاعدة ... -> declaration هي التي نحدد بها شكل التصريح عن المتغيرات, ولدينا ident تعطينا إما متغير أو متغير متبوع بفاصلة للتصريح عن المتغير, هناك يجب أن

نستدعي الدالة ()setup_sym التي تقوم بتثبيت المتغير في جدول المتغيرات و في حالة تكرار التصريح عن نفس المتغير ستعرض خطأ, إسم المتغير القادم من LEX سيكون محفوظ في yylval.Tstr بالنسبة للـ LEX و في $1 بالنسبة للـ BISON.

يبقى الآن التحقق من المتغيرات قبل إسناد قيم لها أو استعمالها حتى, إذا و في كل قاعدة نجد فيها الـ token الخاصة بالمتغير (IDENT) نقوم باستدعاء الدالة ()sym_check للتحقق من المتغير, لاحظ أين يتم هذا:

```
//...
affect:IDENT AFFECT fexpr  {sym_check($1);}
      |IDENT AFFECT sexpr  {sym_check($1);}
      ;
read:IDENT                  {sym_check($1);}
    |IDENT VER read         {sym_check($1);}
    ;
fexpr:REEL
    |ENTIER
    |IDENT                  {sym_check($1);}
    |fexpr PLUS fexpr
    |fexpr MOIN fexpr
    |fexpr MULT fexpr
    |fexpr DIVS fexpr
    |MOIN fexpr %prec NEG
    |fexpr PUIS fexpr
    |PARG fexpr PARD
    ;
//...
```

الأمر واضح, أضفنا des actions sémantiques نستدعي خلالها الدالة sym_check بإعطائها اسم المتغير على شكل بارامتر.

خزن محتوى الملف compalg.y لنرى نتيجة عملنا, احذف الملفات القديمة (compalg.c و compalg.h و lcompalg.c) و قم بتوليد كود المترجم الجديد و هذا بتشغيل الملفات الدفعية BISON_2.bat ثم FLEX_2.bat ثم rename.bat ثم compile.bat لنتحصل على البرنامج النهائي COMPALG.EXE, سنسمي هذه العملية بعملية **توليد المترجم**.

شغل المترجم COMPALG.EXE ليحلل و يكتشف الأخطاء الموجودة على مستوى الملف test.alg و هذه صورة لراسلة الأخطاء:

```
D:\Lex_Yacc\exemples\COMPALG.EXE
Erreur: r est deja definie : ligne 5.
Erreur: x est inconnu : ligne 19.
```

أجل, لقد اكتشف الخطأين الواردين في الـ Algorithme المتعلقين بتكرار التصريح عن المتغيرات و استعمال متغيرات لم يتم التصريح عنها من قبل

بقي لنا أن نكشف عن بقية الأخطاء وهي إسناد قيمة حقيقية لمتغير صحيح أو إسناد سلسلة حرفية لمتغير من نوع char, أكيد هناك المزيد من الأخطاء المعنوية و لكن سنكتفي فقط بما سبق ذكره.

بما أن نوع المتغيرات يهمنا من الآن فصاعدا فإننا سنغير من بنية جدول المتغيرات, إفتح الملف SYMB_TAB.H وغير التركيبة التي تمثل المتغير إلى:

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#define _int          1
#define _float        2
#define _str          3
#define _chr          4
typedef struct sym_node_
{
 char name[56];
 int  type;
 struct sym_node *next;
}sym_node;
//...
```

حددنا أربع ثوابت int_ و float_ و str_ و chr_ تمثل كل من النوع الصحيح و الحقيقي و سلاسل الحروف و الحروف الوحيدة على التوالي, كما قمنا بإضافة حقل آخر للتركيبة التي تحمل معلومات المتغير و هو type و الذي يمثل نوع المتغير.

سنغير أيضا في الدالة put_sym التي تثبت المتغير في الجدول لتصبح:

```
//...
sym_node *put_sym(char *sym_name, int sym_type)
{
 sym_node *ptr;
 ptr=(sym_node*)malloc(sizeof(sym_node));
 strcpy(ptr->name,sym_name);
 ptr->type=sym_type;
 ptr->next=(sym_node*)sym_table;
 sym_table=ptr;
 return ptr;
}
//...
```

واضح ما قمنا به من تغييرات, سنخزن نوع المتغير إضافة إلى اسمه من الآن فصاعدا, سنضيف أيضا دالة جديدة تعيد لنا نوع المتغير المحدد و هي:

```
int get_sym_type(char *sym_name)
{
 sym_node *ptr;
 for(ptr=sym_table;ptr!=NULL;ptr=(sym_node*)ptr->next)
        if(!strcmp(ptr->name,sym_name)) return ptr->type;
 return 0;
}
```

خزن و أغلق الملف SYMB_TAB.H و لننتقل إلى الملف compalg.y.

بما أننا غيرنا في الدالة put_sym الموجودة في الملف SYMB_TAB.H إذ أضفنا إليها نوع المتغير فإنه يجب التغيير في الدالة setup_sym الموجودة في compalg.y لتصبح:

```
//...
void setup_sym(char* sym_name, int type)
{
 sym_node *sym;
 sym=get_sym(sym_name);
 if(sym==NULL) put_sym(sym_name, type);
 else
 {errors++;printf("Erreur: %s est deja definie : ligne
%d.\n",sym_name,line);}
}
//...
```

عند التصريح عن متغير جديد يجب أن نخزن نوعه إضافة إلى اسمه حتى نتمكن لاحقا من التحقق من توافق الأنواع أثنا إجراء عمليات الإسناد, سنضيف متغير جديد current_type نخزن فيه مؤقتا نوع آخر تصريح عن المتغيرات:

```
//...
int errors=0;
int current_type;
//...
```

ولنغير في كل الإستدعاءات للدالة setup_sym:

```
//...
declaration:
           |decl_type  ident  POINT_VER declaration
           ;
//...
ident:IDENT                 {setup_sym($1, current_type);}
     |IDENT VER ident       {setup_sym($1, current_type);}
     ;
//...
```

إذا أصبحا نخزن نوع المتغير مع اسمه, لكن يجب علينا تحديث قيمة current_type في كل مرة يتم التصريح عن نوع جديد, هذا يتم على مستوى هذه القاعدة:

```
//...
decl_type:DEC_ENTIER     {current_type = _int;}
         |DEC_REEL       {current_type = _float;}
         |DEC_CARA       {current_type = _chr;}
         |DEC_CHAINE     {current_type = _str;}
         ;
//...
```

بعد هذا نستطيع التأكد من الأنواع أثناء القيام بالإسناد و هذا يتم على مستوى هذه القاعدة التي تحدد النحو الخاص بالإسناد:

```
affect:IDENT AFFECT fexpr
      |IDENT AFFECT sexpr
      ;
fexpr:REEL
    |ENTIER
    |IDENT
    |fexpr PLUS fexpr
    |fexpr MOIN fexpr
```

```
     |fexpr MULT fexpr
     |fexpr DIVS fexpr
     |MOIN fexpr %prec NEG
     |fexpr PUIS fexpr
     |PARG fexpr PARD
     ;
sexpr:CHAINE
     |CARA
     ;
```

نحن لم نغير أي شيء في الكود السابق, كيف نتحقق؟
لنأخذ مثلا هذه القاعدة:

```
affect : IDENT AFFECT fexpr
```

أثناء إسناد fexpr إلى IDENT يجب التحقق من أن نوع fexpr متوافق مع نوع IDENT وإلا نعرض رسالة خطأ, IDENT هو متغير و لذلك نستطيع الحصول على نوعه من جدول المتغيرات باستعمال الدالة get_sym_type, أما بالنسبة إلى نوع fexpr فيجب علينا الحصول عليه من القاعدة التالية:

```
fexpr : REEL
    | ENTIER
    | IDENT
```

في الحالة الأولى, أي عند fexpr -> reel نرجع النوع _float إلى fexpr, في الحالة الثانية نرجع _int إلى fexpr و في الحالة الثالثة نرجع إلى fexpr نفس نوع المتغير المرفق مع IDENT و هكذا نكمل مع بقية القواعد.. وبما أن fexpr قيمة النوع إذا نوعها يجب أن يكون int و كذلك نفس الشيء بالنسبة لـ sexpr و نحدد ذلك بهذه الإضافة إلى الكود:

```
//...
%union{
 char  Tstr[128];
 int    Tint;
 float  Tfloat;
}
%type  <Tint>   fexpr
%type  <Tint>   sexpr
%token <Tstr>   IDENT
//...
```

إذا أول ما نفعله الآن هو تحديد نوع fexpr و sexpr وهذا بإضافة هذا الكود المفهوم جدا إلى كل من قواعدهما:

```
//...
writed_expr:NEW_LINE
           |fexpr        {/*لن نفعل شيئا هنا*/}
           |sexpr        {/*لن نفعل شيئا هنا  */}
           ;
fexpr:REEL               {$$=_float;}
    |ENTIER              {$$=_int;}
    |IDENT               {
                         if(sym_check($1))
                          $$ = get_sym_type($1);
                         }
    |fexpr PLUS fexpr  {
                        if(($1 == _int) && ($3 == _int))
                         $$ = _int;
                        else
                         $$ = _float;
                       }
    |fexpr MOIN fexpr  {
                        if(($1 == _int) && ($3 == _int))
                         $$ = _int;
                        else
                          $$ = _float;
                        }
    |fexpr MULT fexpr  {
                        if(($1 == _int) && ($3 == _int))
                         $$ = _int;
                        else
                         $$ = _float;
                        }
    |fexpr DIVS fexpr        {$$ = _float;}
    |MOIN fexpr %prec NEG    {$$=$2;}
    |fexpr PUIS fexpr        {$$=$1;}
    |PARG fexpr PARD         {$$=$2;}
    ;
sexpr:CHAINE           {$$=_str;}
    |CARA              {$$=_chr;}
    ;
//...
```

في القاعدة fexpr -> REEL أعدن القيمة _float إلى fexpr باستخدام الكود :

```
$$=_float;
```

في القاعدة IDENT <- fexpr أعدا نفس نوع المتغير IDENT بعد التأطد من أنه مثبت مسبقا في جدول المتغيرات إلى fexpr:

```
if(sym_check($1))
        $$ = get_sym_type($1);
```

في القاعدة fexpr PLUS fexpr <- fexpr فإننا نعيد النوع _int إذا كان كل من fexpr اللذان على اليمين من النوع _int:

```
if(($1 == _int) && ($3 == _int))
    $$ = _int;
else
    $$ = _float;
```

وهكذا نكمل باقي القواعد, الآن استطعنا الحصول على كل من نوع fexpr و sexpr لاستخدامهما في القاعدة المسؤولة عن تصريح المتغيرات, لنفعل ذلك:

```
//...
affect:IDENT AFFECT fexpr    {
                     if(sym_check($1))
                     {
                      int sym_type = get_sym_type($1);
                      if((sym_type==_int) && ($3==_float))
                      {
                       printf("Erreur:imposible de converter (reel
a entier) : ligne %d.\n",line);
                       errors++;
                      }
                      else if((sym_type==_str) && ($3==_float))
                      {
                       printf("Erreur:imposible de converter (reel
a chaine) : ligne %d.\n",line);
                       errors++;
                      }
                      else if((sym_type==_chr) && ($3==_float))
                      {
                       printf("Erreur:imposible de converter (reel
a caractere) : ligne %d.\n",line);
                       errors++;
                      }
                      else if((sym_type==_chr) && ($3==_str))
                      {
                       printf("Erreur:imposible de converter
(chaine a caractere) : ligne %d.\n",line);
                       errors++;
                      }
```

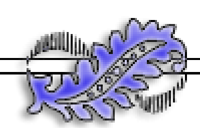
```
                    }
                   }
      |IDENT AFFECT sexpr    {
                    if(sym_check($1))
                    {
                     int sym_type = get_sym_type($1);
                     if((sym_type==_chr) && ($3==_str))
                     {
                      printf("Erreur:imposible de converter
(chaine a caractere) : ligne %d.\n",line);
                      errors++;
                     }
                     else if((sym_type==_int) && ($3==_str))
                     {
                      printf("Erreur:imposible de converter
(chaine a entier) : ligne %d.\n",line);
                      errors++;
                     }
                     else if((sym_type==_float) && ($3==_str))
                     {
                      printf("Erreur:imposible de converter
(chaine a reel) : ligne %d.\n",line);
                      errors++;
                     }
                    }
                   }
      ;
//...
```

مثلا, في القاعدة affect -> IDENT AFFECT fexpr تحققنا أولا من وجود المتغير IDENT باستخدام sym_check($1), إذا كان موجودا نصرح عن متغير جديد باسم sym_type و نخزن فيه نوع المتغير IDENT باستخدام الدالة get_sym_type و بعد ذلك نبدأ عملية التحقق من توافق نوع IDENT و fexpr,و كمثال إذا كان نوع IDENT هو _int و نوع fexpr هو _float فإننا نعرض رسالة خطأ تقول (لا يمكن التحويل من حقيقي إلى صحيح) و هكذا البقية.

أعد الآن توليد المترجم كما تعلمت سابقا و نفذ برنامج المترجم الجديد COMPALG.EXE لتحصل على هذه النتيجة:

```
D:\Lex_Yacc\exemples\COMPALG.EXE
Erreur: r est deja definie : ligne 5.
Erreur:imposible de converter (chaine a caractere) : ligne 12.
Erreur:imposible de converter (chaine a caractere) : ligne 13.
Erreur:imposible de converter (reel a entier) : ligne 17.
Erreur:imposible de converter (reel a entier) : ligne 18.
Erreur: x est inconnu : ligne 19.
```

لقد تمكننا الآن من الكشف عن بقية الأخطاء المعنوية و هذا يكفي الآن بالنسبة لمرحلة التحليل المعنوي أو L'analyse sémantique, المرحلة التالية هي توليد الكود (Génération du code).



توليد الكود, هي عملية صعبة نوعا ما مقارنة مع ما سبقها, سنحاول هنا أن نولد كود assembleur فقط وهو الأقرب إلى لغة الآلة و أيضا سنعتمد طريقة بسيطة في توليد الكود و لن يكون هناك linking (édition de liens) لأننا لن نستورد أي دوال من أي مكاتب.

سنتعامل مع الـ Assembleur 16 bit فقط, لن نحتاج إلى Assembleur 32 bit و هذا لأننا نولد الكود النهائي هنا و سنحاول فيه أن نعتمد فقط على المقاطعات (les interruptions) و لن نتعامل مع الدوال الجاهزة أو les appels système إذ أن الكود يجب أن يكون على أبسط شكل و أقرب شكل إلى لغة الآلة, هذا مثال لكود assembleur يقوم بطباعة الجملة hello world على الشاشة:

```
.MODEL  small
.stack  100h

.data
        msg         db " Hello, World!",13,10,"$"
.code
start:
        mov         ax,@data
        mov         ds,ax

        mov         dx,offset msg
        mov         ah,9
        int         21h         ; المقاطعة التي ستعرض محتوى العنوان
الموجود في دي اكس على الشاشة

        mov         ax,0C07h
        int         21h         ;  الانتظار إلى أن يتم ضغط مفتاح من
لوحة المفاتيح

        mov         ax, 4C00h
        int         21h
end start
```

إذا أردت ترجمته استعمل أي Assembleur كـ WinAsm Studio مثلا, لتفهم بقية الكتاب يجب أن تكون على إطلاع و لو قليل بلغة التجميع مثلي.

أنشي ملف جديد باسم CODE_GEN.H لنكتب فيه بعض الدوال التي تمكننا من كتابة الكود المولد في ملف من نوع *.asm:

```
#include<stdio.h>

char data_section[1024];    //قسم التصريحات
char code_section[8192];    //قسم الكود
FILE *fcode;
              //مقبض للملف الذي سيحمل الكود الناتج
void Init_Code(char *file) // دالة تمكننا من إنشاء ملف جديد ليكون بمثابة نتيجة لترجمة الألغوريتم
{
 fcode = fopen(file, "wb");
 strcpy(data_section, "");
 strcpy(code_section, "");
}

void Add_Data(char *data)    //إضافة تصريح
{
 strcat(data_section, data);
}

void Add_Code(char *code)  //إضافة تعليمة
{
 strcat(code_section, code);
}

void Dispose_Code() // تقوم بإغلاق الملف و إتمام العملية حين ننهي توليد الكود
{
 fprintf(fcode, ".MODEL  small\r\n.stack    100h\r\n.DATA\r\n");
 fprintf(fcode, "%s", data_section);
 fprintf(fcode, ".CODE\r\nstart:\r\n");
 fprintf(fcode, "mov ax,@data\r\n");
 fprintf(fcode, "mov ds,ax\r\n");
 fprintf(fcode, "%s", code_section);
 fprintf(fcode, "end start\r\n");
 fclose(fcode);
}
```

أغلق الملف CODE_GEN.H مع حفظ التغييرات.

إذا إفتح الملف compalg.y و لنبدأ بهذا التغيير بما أننا سنستعمل دوال الملف CODE_GEN.H:

```
//...
#include "d:\lex_yacc\exemples\SYMB_TAB.H"
#include "d:\lex_yacc\exemples\CODE_GEN.H"
//...
```

يجب علينا تهيئة الملف الذي سنكتب فيه الكود المولد و هذا على مستوى الدالة الرئيسية:

```
//...
main(int argc,char *argv[])
{
clrscr();
if((yyin=fopen("d:\\lex_yacc\\exemples\\test.alg","r"))==NULL)
{
 printf("test.alg not found !\n");
 getch();
 return;
}
Init_Code("d:\\lex_yacc\\exemples\\test.asm");
yyparse();
Dispose_Code();
if(!errors) printf("Ok");
getch();
return;
}
//...
```

آه, تذكرت, الملف test.alg كنا قد كتبنا فيه أكواد خاطئة معنويا أثناء برمجة المحلل المعنوي, عد إلى الملف test.alg و أكتب فيه هذا الكود الصحيح ليكون فأر التجربة:

```
algorithme alg
entier resultat,a;
reel r;
caractere car;
chaine s;
debut
ecrire "Entrer a = ";
lire a;
lire r;
lire s;
s<-s;
s<-"0123456789";
ecrire "c";
```

```
car<-"a";
resultat<-5;
a<-resultat*10+5;
ecrire !,"a = ",a;
lire a;
fin.
```

الآن, أعد توليد المترجم COMPALG.EXE و شغله ليقوم بترجمة test.alg و ستلاحظ أنه سيولد لنا الملف الجديد TEST.ASM الذي يحمل كود التجميع, و بما أن كل ما ولدناه إلى الآن هو رأس الكود فسيكون محتواه كالآتي:

```
.MODEL small
.stack 100h
.DATA
.CODE
start:
mov       ax,@data
mov       ds,ax
end start
```

الآن اتضحت لك الفكرة مائة في المائة, لنواصل توليد بقية الكود.

في حالة التصريح عن المتغيرات فإن ذلك يندرج تحت القاعدة التالية:

```
ident:IDENT                  {setup_sym($1, current_type);}
     |IDENT VER ident        {setup_sym($1, current_type);}
     ;
```

بدأنا نقترب من إكمال هذه المرحلة, طبعا لا, مازلنا بعيدين كل البعد و لكن الفكرة تتضح أكثر فأكثر و لهذا أنهي هذا الكتاب هنا و لتكمل البقية بنفسك...أمزح فقط(أردت الهروب قبل أن تزداد المسألة صعوبة..), لا يهم, لنكمل.

التصريح عن المتغيرات, بما أنني مبتدأ جدا في البرمجة بلغة التجميع فسأتعامل مع النوع الحقيقي على أنه صحيح أما سلاسل الحروف و الحروف فسهل التعامل معها, لزيادة الفهم لاحظ هذا التصريح وإلى ماذا سنحوله:

```
entier a;
reel r;
caractere car;
chaine s;
```

→

```
a       dw      6  dup(?)
r       dw      6  dup(?)
car     db      ?
s       db      56  dup(' ')
```

قد لا يكون ذلك التصريح المثالي بالنسبة للغة التجميع و لكن تلك حدودي حاليا, إذا كنت تستطيع فعل ما هو أفضل من ذلك فافعل, نتجه الآن إلى compalg.y لنفعل ذلك:

```
//...
ident:IDENT          {
                     setup_sym($1, current_type);
                     if((current_type == _int) ||
                        (current_type == _float))
                     {
                      Add_Data($1);
                      Add_Data("\tdw\t6 dup(?)\r\n");
                     }
                     else if(current_type == _chr)
                     {
                      Add_Data($1);
                      Add_Data("\tdb\t?\r\n");
                     }
                     else if(current_type == _str)
                     {
                      Add_Data($1);
                      Add_Data("\tdb\t56 dup(‘ ’)\r\n");
                     }
                    }
     |IDENT VER ident        {
                     setup_sym($1, current_type);
                     if((current_type == _int) ||
                       (current_type == _float))
                     {
                      Add_Data($1);
                      Add_Data("\tdw\t6 dup(?)\r\n");
                     }
                     else if(current_type == _chr)
                     {
                      Add_Data($1);
                      Add_Data("\tdb\t?\r\n");
                     }
                     else if(current_type == _str)
                     {
                      Add_Data($1);
                      Add_Data("\tdb\t56 dup(‘ ’)\r\n");
                    }
                    }
     ;
//...
```

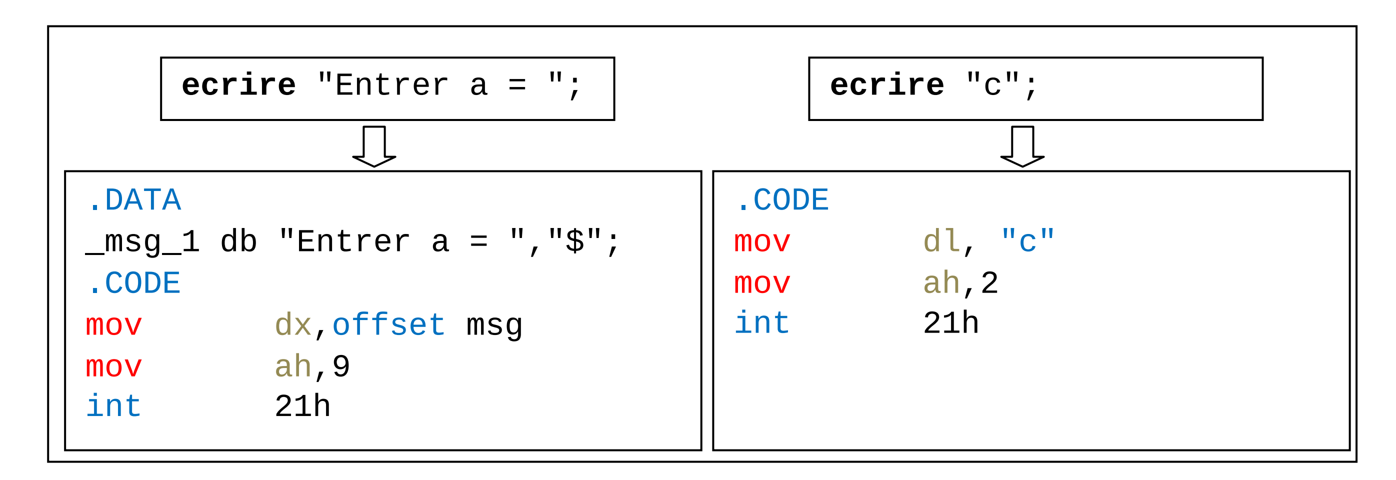
جميل, أعد توليد المترجم ثم شغل البرنامج COMPALG.EXE لينتج لنا الملف TEST.ASM بالمحتوى التالي:

```
.MODEL  small
.stack  100h
.DATA
a       dw6 dup(?)
resultat  dw   6 dup(?)
r       dw6 dup(?)
car     db?
s       db56 dup(' ')
.CODE
start:
mov       ax,@data
mov       ds,ax
end start
```

التصريح عن المتغيرات كان سهلا, أرجو أن يكون الباقي سهلا أيضا أو أسهل.

لنبدأ بتوليد الكود الخاص بـ lire و ecrire, سنبدأ بـ ecrire لأنها تبدو أسهل و بما أنها تخرج نصوص على الشاشة فإننا سنستخدم المقاطعة int 21h, لنرى.

الدالة ecrire قد تأخذ على شكل بارامتر إما متغير أو سلسلة حرفية, إذا كانت سلسلة حرفية فيجب أن نصرح عنها أولا في قسم البيانات ثم نخرجها إلى الشاشة, مثلا:



```
ecrire "Entrer a = ";
.DATA
_msg_1 db "Entrer a = ","$";
.CODE
mov       dx,offset msg
mov       ah,9
int       21h
```

```
ecrire "c";
.CODE
mov       dl,  "c"
mov       ah,2
int       21h
```

لنثبت هذا قبل أن ننتقل إلى غيره, ولكن, عندما يرجع LEX التوكن CARA أو CHAINE يجب أن يرفق معهما قيمة السلسلة الحرفية و لهذا إفتح الملف lcompalg.l:

```
//...
"\"".."\""              {
                         strcpy(yylval.Tstr,yytext);
                         return(CARA);
```

```
                          }
"\""(.)+"\""               {
                           strcpy(yylval.Tstr,yytext);
                           return(CHAINE);
                          }
//...
```

إذا كان بارامتر الدالة ecrire عبارة عن سلسلة حرفية فإننا سنحتاج إلى التصريح عنه و كأنه متغير عادي, لذلك, سنعطي لتلك السلسلة من المتغيرات هذا الشكل _msgX_ حيث سيتغير X من 0 إلى 9, لذلك نعود إلى الملف CODE_GEN.H و لنصرح عن المتغير data_counter ليكون بمثابة عداد لقيمة X و كذلك سنصرح عن متغير مؤقت strtemp وهو عبارة عن مؤشر لسلسلة حرفية قد نحتاجها لحفظ بعض العناوين بشكل مؤقت, كذلك سنحتاج إلى متغير نحدد فيه ما إذا كانت التعليمة الأخيرة أهي ecrire أو إسناد لأنهما يتقاطعان في نفس القاعدة fexpr و sexpr:

```
//...
#include<stdlib.h>
#define _WRITE          2
#define _AFFECT         3

char data_section[1024];
char code_section[8192];
FILE *fcode;
int data_counter = 0;
char *strtemp;
int current_op;
//...
```

الآن لنعرف دالة جديدة تقوم بإرجاع قيمة جديدة للمتغير _msgX_:

```
//...
char* GenStrIdent()
{
 char *temp = (char*)malloc(sizeof(char)*8);
 char string[3];
 itoa(data_counter++, string, 10);
 sprintf(temp, "_msg%s_", string);
 return temp;
}
//...
```

أغلق و احفظ الملف CODE_GEN.H و نعد إلى الملف compalg.y و بالتحديد إلى القاعدة المحدد لشكل الدالة ecrire:

```
//...
%token <Tstr>   CHAINE
%token <Tstr>   CARA
//...
write:writed_expr
     |writed_expr VER write
     ;
writed_expr:NEW_LINE
           |{current_op = _WRITE;}fexpr     {}
           |{current_op = _WRITE;}sexpr     {}
           ;
//...
sexpr:CHAINE         {
                      $$=_str;
                      if(current_op == _WRITE)
                      {
                       strtemp = GenStrIdent();
                       Add_Data(strtemp);
                       Add_Data("\tdb\t");
                       Add_Data($1);
                       Add_Data(",\"$\"\r\n");
                       Add_Code("\r\n;ecrire ");
                       Add_Code($1);
                       Add_Code("\r\n");
                       Add_Code("mov dx, offset ");
                       Add_Code(strtemp);
                       Add_Code("\r\n");
                       Add_Code("mov ah, 9\r\n");
                       Add_Code("int 21h\r\n");
                      }
                     }
    |CARA            {
                      $$=_chr;
                      if(current_op == _WRITE)
                      {
                       Add_Code("\r\n;ecrire ");
                       Add_Code($1);
                       Add_Code("\r\n");
                       Add_Code("mov dl, ");
                       Add_Code($1 );
                       Add_Code("\r\n");
```

```
                    Add_Code("mov ah, 2\r\n");
                    Add_Code("int 21h\r\n");
                   }
                  }
     ;
//...
```

في بداية القاعدة writed_expr حددنا نوع التعليمة التالية و التي ستكون WRITE_ باستعمال {current_op = _WRITE;}, يجب الآن أن نحدد متى ستكون التعليمة التالية غير ذلك و بالتحديد متى ستكون AFFECT_ ؟ هذا يتم قبل بدأ التحقق من القاعدة ... <- affect و لذلك سنضيف قاعدة مساعدة في بدايتها و نسميها er_ :

```
//...
affect:_er IDENT AFFECT fexpr {
                    if(sym_check($2))
                    {
                     int sym_type = get_sym_type($2);
                     if((sym_type==_int) && ($4==_float))
                     {
                      printf("Erreur:imposible de converter (reel
a entier) : ligne %d.\n",line);
                      errors++;
                     }
                     else if((sym_type==_str) && ($4==_float))
                     {
                      printf("Erreur:imposible de converter (reel
a chaine) : ligne %d.\n",line);
                      errors++;
                     }
                     else if((sym_type==_chr) && ($4==_float))
                     {
                      printf("Erreur:imposible de converter (reel
a caractere) : ligne %d.\n",line);
                      errors++;
                     }
                     else if((sym_type==_chr) && ($4==_str))
                     {
                      printf("Erreur:imposible de converter
(chaine a caractere) : ligne %d.\n",line);
                      errors++;
                     }
                    }
                   }
      |_er IDENT AFFECT sexpr {
```

```
                     if(sym_check($2))
                     {
                      int sym_type = get_sym_type($2);
                      if((sym_type==_chr) && ($4==_str))
                      {
                       printf("Erreur:imposible de converter
(chaine a caractere) : ligne %d.\n",line);
                       errors++;
                      }
                      else if((sym_type==_int) && ($4==_str))
                      {
                       printf("Erreur:imposible de converter
(chaine a entier) : ligne %d.\n",line);
                       errors++;
                      }
                      else if((sym_type==_float) && ($4==_str))
                      {
                       printf("Erreur:imposible de converter
(chaine a reel) : ligne %d.\n",line);
                       errors++;
                      }
                     }
                    }
      ;
_er:{current_op = _AFFECT;}
    ;
//...
```

الهدف من القاعدة _er هو إعطاء القيمة _AFFECT للمتغير current_op فقط, و بما أننا أضفنا قاعدة جديدة في بداية affect فإن $1 سيصبح $2 و $3 يصبح $4.

أعد توليد المترجم COMPALG.EXE من جديد ثم شغله ليعطيك ملف TEST.ASM نتيجة وهذا محتواه:

```
//...
_msg0_  db"Entrer a = ","$"
_msg1_  db"a = ","$"
.CODE
start:
mov       ax,@data
mov       ds,ax

;ecrire "Entrer a = "
mov dx, offset _msg0_
```

```
mov ah, 9
int 21h

;ecrire "c"
mov dl, "c"
mov ah, 2
int 21h

;ecrire "a = "
mov dx, offset _msg1_
mov ah, 9
int 21h
end start
```

جيد, بقي لنا إخراج المتغيرات كذلك الرمز الخاص ! الذي يعني سطر جديد, بالنسبة لـ ! فسهل, لنضف إلى قسم البيانات متغير جديد و ثابت و لنسمه _NEWLINE_ و بما أنه ثابت فإننا سنضيفه على شكل قطعة ثابتة من الكود النهائي و هذا في الملف CODE_GEN.H:

```
//...
void Dispose_Code()
{
 fprintf(fcode, ".MODEL  small\r\n.stack     100h\r\n.DATA\r\n");
 fprintf(fcode, "_NEWLINE_\tdb\t13,10,\"$\"\r\n");
 fprintf(fcode, "%s", data_section);
 fprintf(fcode, ".CODE\r\nstart:\r\n");
 fprintf(fcode, "mov ax,@data\r\n");
 fprintf(fcode, "mov ds,ax\r\n");
 fprintf(fcode, "%s", code_section);
 fprintf(fcode, "end start\r\n");
 fclose(fcode);
}
//...
```

عودة إلى compalg.y و لنكمل ذلك:

```
//...
writed_expr:NEW_LINE    {
                         Add_Code("mov dx, offset _NEWLINE_\r\n");
                         Add_Code("mov ah, 9\r\n");
                         Add_Code("int 21h\r\n");
                        }
           |{current_op = _WRITE;}fexpr     {}
           |{current_op = _WRITE;}sexpr     {}
           ;
```

```
//...
```

بالنسبة للمتغيرات, إذا كان المتغير حرفي أو سلسلة حرفية فإن ذلك يتم بسهولة و باستعمال المقاطعة 21 مباشرة مع وضع عنوان المتغير في المسجل dx أو dl:

```
//...
fexpr:REEL                {$$=_float;}
     |ENTIER              {$$=_int;}
     |IDENT          {
                      if(sym_check($1))
                      {
                       $$ = get_sym_type($1);
                       Add_Code("\r\n;ecrire ");
                       Add_Code($1);
                       Add_Code("\r\n");
                       if($$ == _str)
                       {
                        Add_Code("mov dx, offset ");
                        Add_Code($1);
                        Add_Code("\r\n");
                         Add_Code("add dx, 2\r\n");
                        Add_Code("mov ah, 9\r\n");
                        Add_Code("int 21h\r\n");
                       }
                       else if($$ == _chr)
                       {
                        Add_Code("mov dl, ");
                        Add_Code($1);
                        Add_Code("\r\n");
                        Add_Code("mov ah, 2\r\n");
                        Add_Code("int 21h\r\n");
                       }
                      }
                     }
//...
```

أما إذا كان المتغير صحيح –لقد إتفقنا على أن نعامل المتغيرات الحقيقية على أنها صحيحة- فسنستعمل هذه القطعة من الكود التي تعرض متغيرا صحيحا على الشاشة:

```
ten  dw 10
zero  equ  0
N db 6 dup(0)

mov dx,0
```

```
mov ax,145 ;العدد أو المتغير الذي نريد طبعه على الشاشة
mov cx,0
lea bx,N
next0:
  div ten
  cmp ax,zero
  jz  ax0
  jmp cont0
ax0:
  cmp dx,0
  jz  end0
cont0:
  add dx,48
  mov [bx],dx
  inc cx
  inc bx
  mov dx,0
  jmp next0
end0:
dec bx
print0:
  mov al,[bx]
  dec bx
  mov ah,0Eh
  int 10h
  loop print0
```

هناك متغيرات ثابتة و هي _TEN_ و _ZERO_ و _N_ و أيضا هنا بعض les étiquettes أو Labels التي يجب أن تتغير مع كل عملية طبع عدد صحيح, مثلا, اللايبل next0 سيصبح في المرة القادمة next1 و هكذا, نعود إلى الملف CODE_GEN.H لنثبت هذا:

```
//...
int write_int_counter = 0;     //عداد اللايبلس
//...
void Dispose_Code()
{
 fprintf(fcode, ".MODEL  small\r\n.stack     100h\r\n.DATA\r\n");
 fprintf(fcode, "_NEWLINE_\tdb\t13,10,\"$\"\r\n");
 fprintf(fcode, "_TEN_\tdw\t10\r\n");
 fprintf(fcode, "_ZERO_\tequ\t0\r\n");
 fprintf(fcode, "_N_\tdb\t6 dup(0)\r\n");
 fprintf(fcode, "%s", data_section);
 fprintf(fcode, ".CODE\r\nstart:\r\n");
 fprintf(fcode, "mov ax,@data\r\n");
```

```
 fprintf(fcode, "mov ds,ax\r\n");
 fprintf(fcode, "%s", code_section);
 fprintf(fcode, "end start\r\n");
 fclose(fcode);
}

char* GenLabel(char *label) //الدالة التي تولد لايبل جديد
{
 char *temp = (char*)malloc(sizeof(char)*16);
 char string[4];
 itoa(write_int_counter, string, 10);
 sprintf(temp, "%s%s", label, string);
 return temp;
}

void GenLabelIncCounter()
{
 write_int_counter++;
}
//...
```

إذا, إذا أردنا الحصول على next0 فإننا نستدعي الدالة:

```
GenLabel("next") ;
```

و لننتقل إلى المستوى التالي أو next1 نستدعي الدالة GenLabelIncCounter.

عد إلى الملف compalg.y:

```
//...
fexpr:REEL          {$$=_float;}
    |ENTIER         {$$=_int;}
    |IDENT          {
                     if(sym_check($1))
                     {
                      $$ = get_sym_type($1);
                      if(current_op == _WRITE)
                      {
                       Add_Code("\r\n;ecrire ");
                       Add_Code($1);
                       Add_Code("\r\n");
                       if($$ == _str)
                       {
                        Add_Code("mov dx, offset ");
```

```
 Add_Code($1);
 Add_Code("\r\n");
 Add_Code("mov ah, 9\r\n");
 Add_Code("int 21h\r\n");
}
else if($$ == _chr)
{
 Add_Code("mov dl, ");
 Add_Code($1);
 Add_Code("\r\n");
 Add_Code("mov ah, 2\r\n");
 Add_Code("int 21h\r\n");
}
else if(($$ == _float) || ($$ == _int))
{
 Add_Code("mov dx,0\r\n");
 Add_Code("mov ax,");
 Add_Code($1);
 Add_Code("\r\nmov cx,0\r\n");
 Add_Code("lea bx,_N_\r\n");
 Add_Code(GenLabel("next"));
 Add_Code(":\r\n");
 Add_Code("\tdiv _TEN_\r\n");
 Add_Code("\tcmp ax,_ZERO_\r\n");
 Add_Code("\tjz  ");
 Add_Code(GenLabel("ax"));
 Add_Code("\r\n");
 Add_Code("\tjmp ");
 Add_Code(GenLabel("cont"));
 Add_Code("\r\n");
 Add_Code(GenLabel("ax"));
 Add_Code(":\r\n");
 Add_Code("\tcmp dx,0\r\n");
 Add_Code("\tjz  ");
 Add_Code(GenLabel("end"));
 Add_Code("\r\n");
 Add_Code(GenLabel("cont"));
 Add_Code(":\r\n");
 Add_Code("\tadd dx,48\r\n");
 Add_Code("\tmov [bx],dx\r\n");
 Add_Code("\tinc cx\r\n");
 Add_Code("\tinc bx\r\n");
 Add_Code("\tmov dx,0\r\n");
 Add_Code("\tjmp  ");
```

```
                        Add_Code(GenLabel("next"));
                        Add_Code("\r\n");
                        Add_Code(GenLabel("end"));
                        Add_Code(":\r\n");
                        Add_Code("\tdec bx\r\n");
                        Add_Code(GenLabel("print"));
                        Add_Code(":\r\n");
                        Add_Code("\tmov al,[bx]\r\n");
                        Add_Code("\tdec bx\r\n");
                        Add_Code("\tmov ah,0Eh\r\n");
                        Add_Code("\tint 10h\r\n");
                        Add_Code("\tloop ");
                        Add_Code(GenLabel("print"));
                        Add_Code("\r\n");
                        GenLabelIncCounter();
                       }
                      }
                     }
                    }
//...
```

ما زالت هناك حالة أخيرة فيما يخص الدالة ecrire و هي الحالة التي نمرر لها عدد صحيح أو حقيقي مباشرة كبارامتر, مثلا ; ecrire 114, طبعا سنستعمل القاعدة fexpr ->REEL و fexpr: ENTIER, أما بالنسبة للكود فهو نفسه كود طبع المتغيرات الصحيحة مع تغيير التعليمة mov ax, _var_name_ إلى mov ax,$1, أما بالنسبة للقيمة الحقيقية فإننا نحولها إلى قيمة حقيقة أولا باستعمال ; int i = (int)$1 مثلا,لكن و بما أننا سنستعمل نفس الكود لمرتين أخريين فإننا سننشئ دالة جديدة اختصارا لمساحة الكود, لتكن هذه الدالة WriteNum_Code في الملف CODE_GEN.H و ستأخذ على شكل بارامتر اسم المتغير أو القيمة التي سنطبعها, لاحظ:

```
//...
char strtemp1[12];        // متغير مؤقت سنحتاجه لحفظ سلاسل الحروف
مؤقتا
//...
void WriteNum_Code(char *value)
{
 Add_Code("mov dx,0\r\n");
 Add_Code("mov ax,");
 Add_Code(value);
 Add_Code("\r\nmov cx,0\r\n");
 Add_Code("lea bx,_N_\r\n");
 Add_Code(GenLabel("next"));
 Add_Code(":\r\n");
```

```
 Add_Code("\tdiv _TEN_\r\n");
 Add_Code("\tcmp ax,_ZERO_\r\n");
 Add_Code("\tjz  ");
 Add_Code(GenLabel("ax"));
 Add_Code("\r\n");
 Add_Code("\tjmp ");
 Add_Code(GenLabel("cont"));
 Add_Code("\r\n");
 Add_Code(GenLabel("ax"));
 Add_Code(":\r\n");
 Add_Code("\tcmp dx,0\r\n");
 Add_Code("\tjz  ");
 Add_Code(GenLabel("end"));
 Add_Code("\r\n");
 Add_Code(GenLabel("cont"));
 Add_Code(":\r\n");
 Add_Code("\tadd dx,48\r\n");
 Add_Code("\tmov [bx],dx\r\n");
 Add_Code("\tinc cx\r\n");
 Add_Code("\tinc bx\r\n");
 Add_Code("\tmov dx,0\r\n");
 Add_Code("\tjmp  ");
 Add_Code(GenLabel("next"));
 Add_Code("\r\n");
 Add_Code(GenLabel("end"));
 Add_Code(":\r\n");
 Add_Code("\tdec bx\r\n");
 Add_Code(GenLabel("print"));
 Add_Code(":\r\n");
 Add_Code("\tmov al,[bx]\r\n");
 Add_Code("\tdec bx\r\n");
 Add_Code("\tmov ah,0Eh\r\n");
 Add_Code("\tint 10h\r\n");
 Add_Code("\tloop ");
 Add_Code(GenLabel("print"));
 Add_Code("\r\n");
 GenLabelIncCounter();
}
//...
```

بالنسبة للملف compalg.y سنجري هذه التغييرات:

```
//...
fexpr:REEL          {
```

```
                         $$=_float;
                         if(current_op == _WRITE)
                         {
                          sprintf(strtemp1, "%d", (int)$1);
                          Add_Code("\r\n;ecrire ");
                          Add_Code(strtemp1);
                          Add_Code("\r\n");
                          WriteNum_Code(strtemp1);
                         }
                        }
     |ENTIER            {
                         $$=_int;
                         if(current_op == _WRITE)
                         {
                          sprintf(strtemp1, "%d", $1);
                          Add_Code("\r\n;ecrire ");
                          Add_Code(strtemp1);
                          Add_Code("\r\n");
                          WriteNum_Code(strtemp1);
                         }
                        }
//...
```

و بما أننا استعملنا $1 و هي القيمة التي يجب على LEX إعادتها مع التوكن ENTIER أو REEL فيجب التصريح عن نوع كل من القيم المرفقة مع REEL و ENTIER:

```
//...
%token <Tint>   ENTIER
%token <Tfloat>   REEL
//...
```

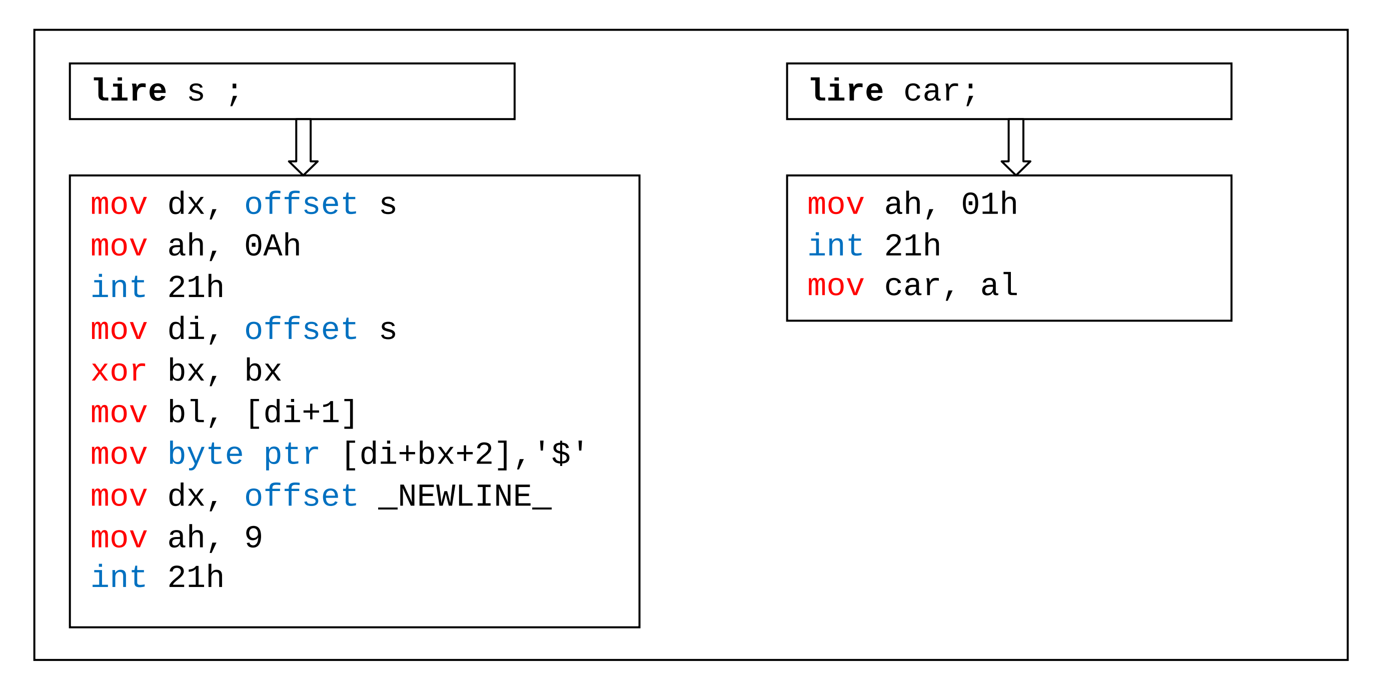
ولنعد إلى الملف lcompalg.l لنعيد القيم مع التوكنز:

```
//...
{reel}           {
                   yylval.Tfloat = atof(yytext);
                   return(REEL);
                 }
{entier}         {
                   yylval.Tint = atoi(yytext);
                   return(ENTIER);
                 }
//...
```

و بهذا ننهي توليد الكود الخاص بالدالة ecrire و إذا أردت رؤية النتيجة الحالية للمترجم فأعد توليده و شغله و افتح الملف test.asm:

```
.MODEL  small
.stack  100h
.DATA
_NEWLINE_ db   13,10,"$"
_TEN_   dw10
_ZERO_  equ    0
_N_     db6 dup(0)
;...
.CODE
start:
;...
;ecrire a
mov dx,0
mov ax,a
mov cx,0
lea bx,_N_
next0:
        div _TEN_
        cmp ax,_ZERO_
        jz  ax0
        jmp cont0
ax0:
        cmp dx,0
        jz  end0
cont0:
        add dx,48
        mov [bx],dx
        inc cx
        inc bx
        mov dx,0
        jmp  next0
end0:
        dec bx
print0:
        mov al,[bx]
        dec bx
        mov ah,0Eh
        int 10h
        loop print0
end start
```

ننتقل الآن إلى الدالة lire و التي ستأخذ على شكل بارامتر إما متغير حرفي أو متغير من نوع سلسلة حرفية أو متغير صحيح أو حقيقي, بالنسبة للمتغيرات الحرفية فالأمر بسيط نوعا ما:



طبعا ذلك على مستوى القاعدة read:

```
//...
read:IDENT             {
                         if(sym_check($1))
                         {
                           Add_Code("\r\n;lire ");
                           Add_Code($1);
                           Add_Code("\r\n");
                           if(get_sym_type($1) == _chr)
                           {
                             Add_Code("mov ah, 01h\r\n");
                             Add_Code("int 21h\r\n");
                             Add_Code("mov ");
                             Add_Code($1);
                             Add_Code(", al\r\n");
                           }
                           else if(get_sym_type($1) == _str)
                           {
                             Add_Code("mov dx, offset ");
                             Add_Code($1);
                             Add_Code("\r\n");
                             Add_Code("mov ah, 0Ah\r\n");
                             Add_Code("int 21h\r\n");
```

```
                        Add_Code("mov di, offset ");
                        Add_Code($1);
                        Add_Code("\r\n");
                        Add_Code("xor bx, bx\r\n");
                        Add_Code("mov bl, [di+1]\r\n");
                        Add_Code("mov byte ptr [di+bx+2],'$'\r\n");
                        Add_Code("mov dx, offset _NEWLINE_\r\n");
                        Add_Code("mov ah, 9\r\n");
                        Add_Code("int 21h\r\n");
                       }
                      }
                     }
    |IDENT VER read       {
                      if(sym_check($1))
                      {
                       Add_Code("\r\n;lire ");
                       Add_Code($1);
                       Add_Code("\r\n");
                       if(get_sym_type($1) == _chr)
                       {
                        Add_Code("mov ah, 01h\r\n");
                        Add_Code("int 21h\r\n");
                        Add_Code("mov ");
                        Add_Code($1);
                        Add_Code(", al\r\n");
                       }
                       else if(get_sym_type($1) == _str)
                       {
                        Add_Code("mov dx, offset ");
                        Add_Code($1);
                        Add_Code("\r\n");
                        Add_Code("mov ah, 0Ah\r\n");
                        Add_Code("int 21h\r\n");
                        Add_Code("mov di, offset ");
                        Add_Code($1);
                        Add_Code("\r\n");
                        Add_Code("xor bx, bx\r\n");
                        Add_Code("mov bl, [di+1]\r\n");
                        Add_Code("mov byte ptr [di+bx+2],'$'\r\n");
                        Add_Code("mov dx, offset _NEWLINE_\r\n");
                        Add_Code("mov ah, 9\r\n");
                        Add_Code("int 21h\r\n");
                       }
                      }
```

```
                             }
    ;
//...
```

ماذا عن قراءة القيم الصحيحة و الحقيقية؟ سنستعمل هذه القطعة من الكود:

```
_ENTER_ equ 13

mov cx,0
next0:
  mov ah,00h
  int 16h
  mov ah,0eh
  int 10h
  cmp al,_ENTER_
  je  end0
  cmp al,'0'
  jb  end0
  cmp al,'9'
  ja  end0
  push ax
  mov ax,cx
  mul _TEN_
  mov cx,ax
  pop ax
  mov ah,0
  sub al,48
  add cx,ax
  jmp next0
end0:
mov a, cx   //المقروؤة القيمة فيه سنحفظ الذي المتغير هو a
```

سنحتاج أولا إلى نصرح عن الثابت _ENTER_ الذي يمثل قيمة الزر ENTER (13) في قسم البيانات و هذا على مستوى الملف CODE_GEN.H:

```
//...
void Dispose_Code()
{
 fprintf(fcode, ".MODEL  small\r\n.stack     100h\r\n.DATA\r\n");
 fprintf(fcode, "_NEWLINE_\tdb\t13,10,\"$\"\r\n");
 fprintf(fcode, "_TEN_\tdw\t10\r\n");
 fprintf(fcode, "_ZERO_\tequ\t0\r\n");
 fprintf(fcode, "_N_\tdb\t6 dup(0)\r\n");
 fprintf(fcode, "_ENTER_\tequ\t13\r\n");
```

```
 fprintf(fcode, "%s", data_section);
 fprintf(fcode, ".CODE\r\nstart:\r\n");
 fprintf(fcode, "mov ax,@data\r\n");
 fprintf(fcode, "mov ds,ax\r\n");
 fprintf(fcode, "%s", code_section);
 fprintf(fcode, "\r\nmov ax,0C07h\r\n");
 fprintf(fcode, "int 21h\r\n");
 fprintf(fcode, "\r\nmov ax, 4C00h\r\n");
 fprintf(fcode, "int 21h\r\n");
 fprintf(fcode, "end start\r\n");
 fclose(fcode);
}
//...
```

وكذلك نضيف الدالة ReadNum_Code المشابهة لـ WriteNum_Code والتي ستقوم بتوليد كود تجميع يقرأ قيمة صحيحة و يسندها للمتغير المعطى على شكل بارامتر:

```
//...
void ReadNum_Code(char *var)
{
 Add_Code("mov cx,0\r\n");
 Add_Code(GenLabel("next"));
 Add_Code(":\r\n");
 Add_Code("\tmov ah, 00h\r\n");
 Add_Code("\tint 16h\r\n");
 Add_Code("\tmov ah, 0Eh\r\n");
 Add_Code("\tint 10h\r\n");
 Add_Code("\tcmp al,_ENTER_\r\n");
 Add_Code("\tje  ");
 Add_Code(GenLabel("end"));
 Add_Code("\r\n");
 Add_Code("\tcmp al,'0'\r\n");
 Add_Code("\tjb  ");
 Add_Code(GenLabel("end"));
 Add_Code("\r\n");
 Add_Code("\tcmp al,'9'\r\n");
 Add_Code("\tja  ");
 Add_Code(GenLabel("end"));
 Add_Code("\r\n");
 Add_Code("\tpush ax\r\n");
 Add_Code("\tmov ax,cx\r\n");
 Add_Code("\tmul _TEN_\r\n");
 Add_Code("\tmov cx,ax\r\n");
 Add_Code("\tpop ax\r\n");
```

```
 Add_Code("\tmov ah,0\r\n");
 Add_Code("\tsub al,48\r\n");
 Add_Code("\tadd cx,ax\r\n");
 Add_Code("\tjmp  ");
 Add_Code(GenLabel("next"));
 Add_Code("\r\n");
 Add_Code(GenLabel("end"));
 Add_Code(":\r\n");
 Add_Code("\tmov ");
 Add_Code(var);
 Add_Code(",cx\r\n");
 GenLabelIncCounter();
}
```

نعود إلى الملف compalg.y:

```
//...
read:IDENT           {
                       //...
                       else if((get_sym_type($1) == _int) ||
                               (get_sym_type($1) == _float))
                       {
                        ReadNum_Code ($1);
                       }
                      }
                     }
    |IDENT VER read {
                       //...
                       else if((get_sym_type($1) == _int) ||
                               (get_sym_type($1) == _float))
                       {
                        ReadNum_Code ($1);
                       }
                      }
                     }
    ;
//...
```

بهذا ننهي توليد الكود الخاص بالدالة lire, بقي فقط توليد الكود الموافق للإسنادات بين المتغيرات و القيم وباقي العمليات الحسابية الرياضية, البداية تكون من القاعدة التالية:

```
affect:_er IDENT AFFECT fexpr
```

نضيف أولا action sémantique بواسطتها نولد تعليق نشير به إلى أن القطعة القادمة من كود التجميع تمثل عملية إسناد:

```
//...
affect:_er IDENT AFFECT {Add_Code("\r\n;");Add_Code($2);Add_Code("
<- ...\r\n");}fexpr {
                    if(sym_check($2))
                    {
                     //...
                     if((sym_type==_int) || (sym_type==_float))
                     {
                      //IDENT نقوم بإسناد النتيجة إلى
                     }
                     else if(sym_type==_chr)
                     {
                      //IDENTنقوم بإسناد النتيجة إلى
                      Add_Code("mov ");
                      Add_Code($2);
                      Add_Code(",al\r\n");
                     }
                    }
                   }
      |_er IDENT AFFECT {Add_Code("\r\n;");Add_Code($2);Add_Code("
<- ...\r\n");}sexpr {
                    if(sym_check($2))
                    {
                     //...
                     if((sym_type==_int) || (sym_type==_float))
                     {
                      //IDENTنقوم بإسناد النتيجة إلى
                     }
                     else if(sym_type==_chr)
                     {
                      //IDENTنقوم بإسناد النتيجة إلى
                      Add_Code("mov ");
                      Add_Code($2);
                      Add_Code(",al\r\n");
                     }
                    }
                   }
      ;
//...
```

بدأنا بالتعامل مع الحروف, إذ سنقوم بإسناد حرف إلى متغير حرفي عن طريق التعليمة:

```
mov car, al
```

حيث car هو اسم المتغير, يبقى الآن أن نضع قيمة الحرف داخل المسجل al:

```
//...
sexpr:CHAINE        {
                     //...
                    }
    |CARA           {
                     //...
                     else if(current_op == _AFFECT)
                     {
                       Add_Code("mov al, ");
                       Add_Code($1);
                       Add_Code("\r\n");
                     }
                    }
    ;
//...
```

ننتقل إلى حالة إسناد سلسلة حرفية إلى متغير, هنا يجب أن نصرح بتلك السلسلة في قسم البيانات على أنها متغير ثم نسندها إلى المتغير باستخدام حلقة, مثلا, إذا كان لدينا هذا الإسناد:

```
s<-"0123456789";
```

سنحولها إلى هذا الكود:

```
_msg2_  db"0123456789","$"

mov di,offset s     ; الحرفية السلسلة إليه سنسند الذي المتغير هو s
mov si,offset _msg2_
mov cx,[si+1]
next0:
mov al,[si]
cmp al,0
jz  end0
mov byte ptr [di],al
inc si
inc di
dec cx
cmp cx,0
je end0
loop next0
end0:
```

نحن سنحتاج إلى أن ننقل إسم المتغير من القاعدة affect إلى القاعدة sexpr و لذلك سنحفظه في المتغير المؤقت المتعدد الإستعمالات strtemp1:

```
//...
affect:_er IDENT AFFECT {Add_Code("\r\n;");Add_Code($2);Add_Code("
<- ...\r\n"); }fexpr          {
                               //...
                              }
      |_er IDENT AFFECT {Add_Code("\r\n;");Add_Code($2);Add_Code("
<- ...\r\n");strcpy(strtemp1,$2);}sexpr{
                                         //...
                                      }
//...
```

أما باقي العمل فيتم على مستوى القاعدة sexpr:CHAINE:

```
//...
sexpr:CHAINE        {
                     //...
                     else if(current_op == _AFFECT)
                     {
                      strtemp = GenStrIdent();
                      Add_Data(strtemp);
                      Add_Data("\tdb\t");
                      Add_Data($1);
                      Add_Data(",\"$\"\r\n");
                      Add_Code("mov di,offset ");
                      Add_Code(strtemp1);
                      Add_Code("\r\n");
                      Add_Code("mov si,offset ");
                      Add_Code(strtemp);
                      Add_Code("\r\n");
                      Add_Code("mov cx,[si+1]\r\n");
                      Add_Code(GenLabel("next"));
                      Add_Code(":\r\n");
                      Add_Code("\tmov al,[si]\r\n");
                      Add_Code("\tcmp al,0\r\n");
                      Add_Code("\tjz  ");
                      Add_Code(GenLabel("end"));
                      Add_Code("\r\n");
                      Add_Code("\tmov byte ptr [di],al\r\n");
                      Add_Code("\tinc si\r\n");
                      Add_Code("\tinc di\r\n");
                      Add_Code("\tdec cx\r\n");
```

```
                          Add_Code("\tcmp cx,0\r\n");
                          Add_Code("\tjz  ");
                          Add_Code(GenLabel("end"));
                          Add_Code("\r\n");
                          Add_Code("\tloop  ");
                          Add_Code(GenLabel("next"));
                          Add_Code("\r\n");
                          Add_Code(GenLabel("end"));
                          Add_Code(":\r\n");
                          GenLabelIncCounter();
                         }
                        }
//...
```

كان ذلك سهلا نوعا ما, بنفس الطريقة سنولد كود إسناد متغير حرفي إلى متغير حرفي, لكن و بما أن كل المتغيرات تتقاطع في القاعدة fexpr -> IDENT فيجب علينا إضافة متغير جديد يحمل قيمة تحدد نوع المتغير الذي سنسند إليه قيمة المتغير IDENT الموجود في القاعدة fexpr -> IDENT, سنسمي هذا المتغير AffectIdentType و سنصرح عنه في بداية الملف compalg.y:

```
%{
#include<conio.h>
#include<math.h>
#include "d:\lex_yacc\exemples\lcompalg.c"
#include "d:\lex_yacc\exemples\SYMB_TAB.H"
#include "d:\lex_yacc\exemples\CODE_GEN.H"
int errors=0;
int current_type;
int AffectIdentType;
//...
```

و سنسند لهذا المتغير نوع المتغير الذي يجب إسناد القيمة له (_int, _float, _str, _chr) و هذا في بداية القاعدة affect:

```
//...
affect:_er IDENT AFFECT {Add_Code("\r\n;");Add_Code($2);Add_Code("
<-
...\r\n");strcpy(strtemp1,$2);AffectIdentType=get_sym_type($2);}fex
pr      {
                                                        //...
                                                       }
      |_er IDENT AFFECT {Add_Code("\r\n;");Add_Code($2);Add_Code("
<-
```

```
...\r\n");strcpy(strtemp1,$2);AffectIdentType=get_sym_type($2);}sex
pr      {
         //...
        }
;
//...
```

أما الباقي فسنكمله على مستوى القاعدة fexpr -> IDENT:

```
//...
fexpr:REEL          {
                     //...
                    }
     |ENTIER        {
                     //...
                    }
     |IDENT          {
                      //...
                     else if(current_op == _AFFECT)
                     {
                      if(AffectIdentType == _str)
                      {
                        Add_Code("mov di,offset ");
                        Add_Code(strtemp1);
                        Add_Code("\r\n");
                        Add_Code("mov si,offset ");
                        Add_Code($1);
                        Add_Code("\r\n");
                        Add_Code("mov cx,[si+1]\r\n");
                        Add_Code(GenLabel("next"));
                        Add_Code(":\r\n");
                        Add_Code("\tmov al,[si]\r\n");
                        Add_Code("\tcmp al,0\r\n");
                        Add_Code("\tjz  ");
                        Add_Code(GenLabel("end"));
                        Add_Code("\r\n");
                        Add_Code("\tmov byte ptr [di],al\r\n");
                        Add_Code("\tinc si\r\n");
                        Add_Code("\tinc di\r\n");
                        Add_Code("\tdec cx\r\n");
                        Add_Code("\tcmp cx,0\r\n");
                        Add_Code("\tjz  ");
                        Add_Code(GenLabel("end"));
                        Add_Code("\r\n");
```

```
                                Add_Code("\tloop  ");
                                Add_Code(GenLabel("next"));
                                Add_Code("\r\n");
                                Add_Code(GenLabel("end"));
                                Add_Code(":\r\n");
                                GenLabelIncCounter();
                               }
                              }
                            }
//...
```

لن ننسى أيضا الحالة التي نسند فيها متغير حرفي إلى متغير حرفي, كل ما سنفعله هو وضع قيمة المتغير الحرفي المراد إسناده في المسجل al:

```
//...
fexpr:REEL          {
                     //...
                    }
    |ENTIER         {
                     //...
                    }
    |IDENT           {
                      //...
                      else if(current_op == _AFFECT)
                      {
                       else if(AffectIdentType == _chr)
                       {
                        Add_Code("mov al, ");
                        Add_Code($1);
                        Add_Code("\r\n");
                       }
                      }
                    }
//...
```

سنتجاهل العمليات التي تتم بين سلاسل الحروف كـ strcpy و strcat و غيرها لأن هدف هذا الكتاب تعليمي و ليس برمجة مترجم من الألف إلى الياء, و هنا ننهي توليد كود التجميع الخاص بإسناد المتغيرات الحرفية و سلاسل الحروف إلى المتغيرات الحرفية.

بقي لنا إنهاء عمليات الإيناد بين المتغيرات الصحيحة و الحقيقية, القاعدة المسؤولة عن الإسناد –كما أسرنا إليها عشرات المرات من قبل- هي:

```
affect:_er IDENT AFFECT fexpr
```

علينا القيام بالحسابات الموجودة في fexpr ثم وضعها داخل المسجل AX و عند إكتمال القاعدة نسند القيمة الموجودة في AX إلى المتغير IDENT بواسطة الكود mov $1, AX, لنكتب هذا الكود أولا لأنه يبدو سهلا:

```
//...
affect:_er IDENT AFFECT {/*…*/}fexpr    {
                     if(sym_check($2))
                     {
                      //...
                      if((sym_type==_int) || (sym_type==_float))
                      {
                       Add_Code("mov ");
                       Add_Code($2);
                       Add_Code(",ax\r\n");
                      }
                      //...
                     }
                    }
//...
```

لننسى الآن أمر المتغير و اسمه فهو لن يهمنا و لنقم ببقية الحسابات الموجودة في fexpr و لنسندها إلى المسجل ax, نحتاج إلى متغير ولنسمه CurrentOp من نوع int سيحمل إما القيمة NULL و التي تعني أنه لا يوجد لدينا أي عملية حاليا و في هذه الخالة سنسند القيمة الخحالية إلى ax مباشرة mov ax, 112, أما إذا كانت قيمة المتغير CurrentOp مثلا PLUS فهذا يعني أن العملية الحالية هي الجمع, أي add ax, 112:

```
//...
int errors=0;
int current_type;
int AffectIdentType;
int CurrentOp = NULL;
//...
```

أين سنحدد القيمة الحالية لـ CurrentOp ؟ طبعا على مستوى القاعدة fexpr:

```
//...
fexpr:REEL           {
                      //...
                     }
     |ENTIER         {
                      //...
                     }
```

```
     |IDENT              {
                          //...
                         }
     |fexpr PLUS {CurrentOp = PLUS;} fexpr        {
                          if(($1 == _int) && ($4 == _int))
                           $$ = _int;
                          else
                           $$ = _float;
                         }
     |fexpr MOIN {CurrentOp = MOIN;} fexpr        {
                          if(($1 == _int) && ($4 == _int))
                           $$ = _int;
                          else
                           $$ = _float;
                         }
     |fexpr MULT {CurrentOp = MULT;} fexpr        {
                          if(($1 == _int) && ($4 == _int))
                           $$ = _int;
                          else
                           $$ = _float;
                         }
     |fexpr DIVS {CurrentOp = DIVS;} fexpr        {$$ = _float;}
     |MOIN fexpr %prec NEG                        {$$=$2;}
     |fexpr PUIS {CurrentOp = PUIS;} fexpr        {$$=$1;}
     |PARG fexpr PARD            {$$=$2;}
     ;
//...
```

أما باقي العمل فنتمه على مستوى القواعد fexpr -> ENTIER و fexpr -> REEL و كذلك fexpr -> IDENT و في نهاية كل قاعدة من القواعد الثلاث يجب إرجاع قيمة CurrentOp إلى NULL:

```
//...
fexpr:REEL               {
                          //...
                          CurrentOp = NULL;
                         }
     |ENTIER             {
                          //...
                          CurrentOp = NULL;
                         }
     |IDENT              {
                          //...
                          CurrentOp = NULL;
```

```
                    }
//...
```

أسهل الحالات بالنسبة لـ CurrentOp هي NULL و PLUS, الكود المحدد لكلتا الحالتين هو:

```
//...
fexpr:REEL          {
                      //...
                      if(CurrentOp == NULL)
                      {
                        Add_Code("mov ax, ");
                        Add_Code(strtemp1);
                        Add_Code("\r\n");
                      }
                      else if(CurrentOp == PLUS)
                      {
                        Add_Code("add ax, ");
                        Add_Code(strtemp1);
                        Add_Code("\r\n");
                      }
                     }
                     CurrentOp = NULL;
                    }
    |ENTIER         {
                     //...
                     else if(current_op == _AFFECT)
                     {
                      sprintf(strtemp1, "%d", (int)$1);
                      if(CurrentOp == NULL)
                      {
                        Add_Code("mov ax, ");
                        Add_Code(strtemp1);
                        Add_Code("\r\n");
                      }
                      else if(CurrentOp == PLUS)
                      {
                        Add_Code("add ax, ");
                        Add_Code(strtemp1);
                        Add_Code("\r\n");
                      }
                     }
                     CurrentOp = NULL;
                    }
    |IDENT          {
```

```
                     if(sym_check($1))
                     {
                      //...
                      else if(current_op == _AFFECT)
                      {
                       //...
                       else if((AffectIdentType == _int) ||
(AffectIdentType == _float))
                       {
                        if(CurrentOp == NULL)
                        {
                         Add_Code("mov ax, ");
                         Add_Code($1);
                         Add_Code("\r\n");
                        }
                        else if(CurrentOp == PLUS)
                        {
                         Add_Code("add ax, ");
                         Add_Code($1);
                         Add_Code("\r\n");
                        }
                       }
                      }
                     }
                     CurrentOp = NULL;
                    }
//...
```

لننتقل إلى العملية MULT, إذا كانت لدينا مثلا العملية 10*6, سنقوم بوضع القيمة 6 داخل المسجل ax, بعد ذلك سنجد أمامنا العملية MULT في المتغير CurrentOp, ماذا نفعل؟ سنطرح القيمة 6 من المسجل AX بإستخدام sub ax,6 بعد ذلك نجري عملية الضرب... سيكون هذا إهدارا واضحا لوقت المعالج مع أني لم أضع وقت المعالج في الحسبان (تجاهلنا عملية تحسين الكود أو Optimisation du code), سنغير في الكود السابق و بالتحديد في الحالة NULL للمتغير CurrentOp.

لن نقوم بوضع القيمة الحالية أو المتغير الحالي داخل المسجل ax من الآن فصاعدا, بل سنقوم بوضعها في متغير مؤقت بإسم TempReg الذي سيكون سلسلة حرفية تحمل إسم المتغير أو القيمة المشطلة لأحد أطراف العملية الحالية, بعد ذلك و إذا صادفتنا عملية جديدة فإننا سنجريها مع محتوى TempReg, أما إذا لم تكن هناك أي عملية جديدة فإننا نضيف TempRegإلى ax في نهاية القاعدة affect و أيضا تصفير المسجل ax في البداية كالآتي:

```
//...
int CurrentOp = NULL;
```

```
char TempReg[16];
//...
affect:_er IDENT AFFECT {/*...*/ Add_Code("mov ax,0\r\n");}fexpr
                    {
                     if(sym_check($2))
                     {
                      //...
                      if((sym_type==_int) || (sym_type==_float))
                      {

                       Add_Code("add ax,");
                       Add_Code(TempReg);
                       Add_Code(",ax\r\n");
                       Add_Code("mov ");
                       Add_Code($2);
                       Add_Code(",ax\r\n");
                      }
                      //...
                     }
                    }
//...
```

و لنغير في الحالة NULL للمتغير CurrentOp في القاعدة fexpr:

```
//...
fexpr:REEL          {
                     //...
                     else if(current_op == _AFFECT)
                     {
                      sprintf(strtemp1, "%d", (int)$1);
                      if(CurrentOp == NULL)
                      {
                       strcpy(TempReg, strtemp1);
                      }
                      //...
                     }
                     CurrentOp = NULL;
                    }
    |ENTIER         {
                     //...
                     else if(current_op == _AFFECT)
                     {
                      sprintf(strtemp1, "%d", (int)$1);
                      if(CurrentOp == NULL)
```

```
                       {
                        strcpy(TempReg, strtemp1);
                       }
                       //...
                      }
                      CurrentOp = NULL;
                     }
     |IDENT              {
                      if(sym_check($1))
                      {
                       //...
                       else if(current_op == _AFFECT)
                       {
                        //...
                        else if((AffectIdentType == _int) ||
(AffectIdentType == _float))
                        {
                         if(CurrentOp == NULL)
                         {
                          strcpy(TempReg, $1);
                         }
                         //...
                        }
                       }
                      }
                      CurrentOp = NULL;
                     }
//...
```

نعود إلى العملية 10*6, الكود الذي يحسبها هو :

```
push ax
mov ax, 6
mov bx, 10
mul bx
mov cx, ax
pop ax
```

إذن النتيجة الأخيرة موجودة في المسجل cx, إذا كانت العملية لن نضيف قيمته إلى المسجل ax إلا إذا تأكدنا أن العملية القادمة هي جمع أو طرح, أما إذا كانت قسمة أو ضرب فإننا سنجريها على النتيجة الحالية الموجودة في المسجل cx و ليس على النتيجة الكلية الموجودة في ax, لذلك نضيف متغيرا جديدا بإسم PrevOp و الذي سيحمل إما القيمة NULL أو قيمة العملية السابقة:

```
//...
int CurrentOp = NULL;
int PrevOp = NULL;
//...
fexpr:REEL          {
                     //...
                     else if(current_op == _AFFECT)
                     {
                      sprintf(strtemp1, "%d", (int)$1);
                      if(CurrentOp == NULL)
                      {
                       strcpy(TempReg, strtemp1);
                      }
                      else if(CurrentOp == PLUS)
                      {
                       if(PrevOp == MULT)
                       {
                        Add_Code("add ax, cx\r\n");
                       }
                       Add_Code("add ax, ");
                       Add_Code(strtemp1);
                       Add_Code("\r\n");
                      }
                      else if(CurrentOp == MULT)
                      {
                       if(PrevOp == MULT)
                       {
                        Add_Code("push ax\r\n");
                        Add_Code("mov ax, cx\r\n");
                        Add_Code("mov bx,");
                        Add_Code(strtemp1);
                        Add_Code("\r\n");
                        Add_Code("mul bx\r\n");
                        Add_Code("mov cx,ax\r\n");
                        Add_Code("pop ax\r\n");
                       }
                       else
                       {
                        Add_Code("push ax\r\n");
                        Add_Code("mov ax,");
                        Add_Code(TempReg);
                        Add_Code("\r\n");
                        Add_Code("mov bx,");
                        Add_Code(strtemp1);
```

```
              Add_Code("\r\n");
              Add_Code("mul bx\r\n");
              Add_Code("mov cx,ax\r\n");
              Add_Code("pop ax\r\n");
             }
            }
           }
           PrevOp = CurrentOp;
           CurrentOp = NULL;
          }
|ENTIER   {
           //...
           else if(current_op == _AFFECT)
           {
            sprintf(strtemp1, "%d", (int)$1);
            if(CurrentOp == NULL)
            {
             strcpy(TempReg, strtemp1);
            }
            else if(CurrentOp == PLUS)
            {
             if(PrevOp == MULT)
             {
              Add_Code("add ax, cx\r\n");
             }
             Add_Code("add ax, ");
             Add_Code(strtemp1);
             Add_Code("\r\n");
            }
            else if(CurrentOp == MULT)
            {
             if(PrevOp == MULT)
             {
              Add_Code("push ax\r\n");
              Add_Code("mov ax, cx\r\n");
              Add_Code("mov bx,");
              Add_Code(strtemp1);
              Add_Code("\r\n");
              Add_Code("mul bx\r\n");
              Add_Code("mov cx,ax\r\n");
              Add_Code("pop ax\r\n");
             }
             else
             {
```

```
                        Add_Code("push ax\r\n");
                        Add_Code("mov ax,");
                        Add_Code(TempReg);
                        Add_Code("\r\n");
                        Add_Code("mov bx,");
                        Add_Code(strtemp1);
                        Add_Code("\r\n");
                        Add_Code("mul bx\r\n");
                        Add_Code("mov cx,ax\r\n");
                        Add_Code("pop ax\r\n");
                       }
                      }
                     }
                     PrevOp = CurrentOp;
                     CurrentOp = NULL;
                    }
     |IDENT         {
                     if(sym_check($1))
                     {
                      //...
                      else if(current_op == _AFFECT)
                      {
                       //...
                       else if((AffectIdentType == _int) ||
(AffectIdentType == _float))
                       {
                        if(CurrentOp == NULL)
                        {
                         strcpy(TempReg, $1);
                        }
                        else if(CurrentOp == PLUS)
                        {
                         if(PrevOp == MULT)
                         {
                          Add_Code("add ax, cx\r\n");
                         }
                         Add_Code("add ax, ");
                         Add_Code($1);
                         Add_Code("\r\n");
                        }
                        else if(CurrentOp == MULT)
                        {
                         if(PrevOp == MULT)
                         {
```

```
                        Add_Code("push ax\r\n");
                        Add_Code("mov ax, cx\r\n");
                        Add_Code("mov bx,");
                        Add_Code(strtemp1);
                        Add_Code("\r\n");
                        Add_Code("mul bx\r\n");
                        Add_Code("mov cx,ax\r\n");
                        Add_Code("pop ax\r\n");
                       }
                       else
                       {
                        Add_Code("push ax\r\n");
                        Add_Code("mov ax,");
                        Add_Code($1);
                        Add_Code("\r\n");
                        Add_Code("mov bx,");
                        Add_Code(strtemp1);
                        Add_Code("\r\n");
                        Add_Code("mul bx\r\n");
                        Add_Code("mov cx,ax\r\n");
                        Add_Code("pop ax\r\n");
                       }
                      }
                     }
                    }
                   }
                   PrevOp = CurrentOp;
                   CurrentOp = NULL;
                  }
//...
```

أثناء توليد كود عملية الضرب لا بد أن نجري إختبارا على المتغير PrevOp, إذا كان محتواه MULT فإن هذا يعني أننا أجرينا عملية ضرب قبل هذه العملية و عملية الضرب الحالية يجب أن تجرى على نتيجة العملية السابقة و الموجودة في المسجل CX, أما إذا كان محتواه غير ذلك فإن هذا يعني أن عملية الضرب الحالية سبقتها عملية جمع أو لاشيء و من هذا فإننا سنجريها على القيمة المحفوظة في المتغير المؤقت TempReg وهكذا.

علينا الآن أن نعود إلى نهاية القاعدة affect وبالتحديد إلى الكود الذي نقوم فيه بإسناد النتيجة الموجودة في المسجل AX إلى المتغير المقصود, لن نضيف القيمة الموجودة في tempReg إلى المسجل AX في كل الحالات, بل فقط في حالة كون عملية الإسناد لا تحوي أي عمليات رياضية في جانبها الأيمن, أي أن قيمة PrevOp مساوية لـ NULL:

```
//...
affect:_er IDENT AFFECT {/*...*/}fexpr {
                    if(sym_check($2))
                    {
                     //...
                     if((sym_type==_int) || (sym_type==_float))
                     {
                      if(PrevOp == NULL)
                      {
                       Add_Code("add ax,");
                       Add_Code(TempReg);
                       Add_Code("\r\n");
                      }
                      Add_Code("mov ");
                      Add_Code($2);
                      Add_Code(",ax\r\n");
                     }
                     //...
                    }
                   }
//...
```

بعد تنفيذ كل تعليمات لغة التجميع يجب علينا إستدعاء المقاطعة 21 مع الدالة 0C07h لتقرأ حرف من لوحة المفاتيح حتى نتمكن من رؤية نتيجة البرنامج, بعد ذلك نستدعي المقاطعة 21 مع الدالة 4C00h وهي مقاطعة الخروج:

```
mov      ax,0C07h       ; Function 0Ch = "FLUSH BUFFER AND READ
                         ;STANDARD INPUT"
int      21h            ; Waits for a key to be pressed.
mov      ax, 4C00h      ; the exit fuction  [4C+no error (00)]
int      21h            ; call DOS interrupt 21h
```

ذلك يتم على مستوى الملف CODE_GEN.H:

```
//...
void Dispose_Code()
{
 fprintf(fcode, ".MODEL  small\r\n.stack    100h\r\n.DATA\r\n");
 fprintf(fcode, "_NEWLINE_\tdb\t13,10,\"$\"\r\n");
 fprintf(fcode, "_TEN_\tdw\t10\r\n");
 fprintf(fcode, "_ZERO_\tequ\t0\r\n");
```

```
 fprintf(fcode, "_N_\tdb\t6 dup(0)\r\n");
 fprintf(fcode, "_ENTER_\tequ\t13\r\n");
 fprintf(fcode, "%s", data_section);
 fprintf(fcode, ".CODE\r\nstart:\r\n");
 fprintf(fcode, "mov ax,@data\r\n");
 fprintf(fcode, "mov ds,ax\r\n");
 fprintf(fcode, "%s", code_section);
 fprintf(fcode, "\r\nmov ax,0C07h\r\n");
 fprintf(fcode, "int 21h\r\n");
 fprintf(fcode, "\r\nmov ax, 4C00h\r\n");
 fprintf(fcode, "int 21h\r\n");
 fprintf(fcode, "end start\r\n");
 fclose(fcode);
}
//...
```

لنجرب ما أنجزناه إلى حد الآن على هذا الألغوريتم:

```
algorithme alg
entier resultat,a;
debut
ecrire "Entrer a = ";
lire a;
resultat<-5*a+10;
a<-resultat;
ecrire !,"a = ",a;
fin.
```

بعد إعادة توليد المترجم COMPALG.EXE و تشغيله سنحصل على الملف TEST.ASM كنتيجة لترجمة الألغوريتم السابق الموجود في الملف test.alg:

```
.MODEL  small
.stack  100h
.DATA
_NEWLINE_ db   13,10,"$"
_TEN_   dw10
_ZERO_  equ     0
_N_     db6 dup(0)
_ENTER_ equ     13
a       dw6 dup(?)
resultat  dw   6 dup(?)
_msg0_  db"Entrer a = ","$"
_msg1_  db"a = ","$"
.CODE
```

```
start:
mov ax,@data
mov ds,ax

;ecrire "Entrer a = "
mov dx, offset _msg0_
mov ah, 9
int 21h

;lire a
mov cx,0
next0:
        mov ah, 00h
        int 16h
        mov ah, 0Eh
        int 10h
        cmp al,_ENTER_
        je  end0
        cmp al,'0'
        jb  end0
        cmp al,'9'
        ja  end0
        push ax
        mov ax,cx
        mul _TEN_
        mov cx,ax
        pop ax
        mov ah,0
        sub al,48
        add cx,ax
        jmp  next0
end0:
        mov a,cx

;resultat <- ...
mov ax,0
push ax
mov ax,a
mov bx,5
mul bx
mov cx,ax
pop ax
add ax, cx
add ax, 10
```

```
mov resultat,ax

;a <- ...
mov ax,0
add ax,resultat
mov a,ax
mov dx, offset _NEWLINE_
mov ah, 9
int 21h

;ecrire "a = "
mov dx, offset _msg1_
mov ah, 9
int 21h

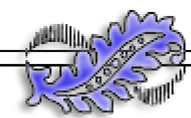
;ecrire a
mov dx,0
mov ax,a
mov cx,0
lea bx,_N_
next1:
        div _TEN_
        cmp ax,_ZERO_
        jz  ax1
        jmp cont1
ax1:
        cmp dx,0
        jz  end1
cont1:
        add dx,48
        mov [bx],dx
        inc cx
        inc bx
        mov dx,0
        jmp  next1
end1:
        dec bx
print1:
        mov al,[bx]
        dec bx
        mov ah,0Eh
        int 10h
        loop print1
```

```
mov ax,0C07h
int 21h

mov ax, 4C00h
int 21h
end start
```

أنسخ الكود و ألصقه في محرر برنامج WinAsm Studios لتكون نتيجة تنفيذ الكود السابق كالتالي:

```
E:\WinAsm\DosExe.exe
Une ou plusieurs pages de codes CON non valides pour ce code de clavier
Entrer a = 2
a = 20
```



أما باقي العمليات من قسمة و طرح و رفع إلى أس فكلها مشابهة لما سبق إنجازه, لذلك نتوقف هنا في هذه المرحلة من الترجمة و في الكتاب ككل مع أني كنت أريد الوصول إلى توليد ملف تنفيذي للكود و لكن... لأسباب عدة أهمها أنني لا أعرف كيف أحول تعليمات لغة التجميع إلى لغة الآلة حاليا.

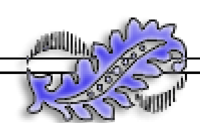
في النهاية أرجو المعذرة للأخطاء التي و إن لم ألاحظ و جودها إلا أنها لا بد أن تكون, فالكمال لله, و إن كانت هناك أخطاء كارثية فأرجو منك مراسلتي لأصححها بإذن الله, أتتركم في رعاية الله و حفظه و السلام عليكم و رحمة الله وبركاته.

لتحميل كود المترجم المنجز في هذا الكتاب إتبع أحد هذه الروابط:

http://www.4shared.com/file/231365377/3f824ec0/exemples.html

http://www.mediafire.com/?zy2eynn1gaj

http://www.snapdrive.net/files/618263/CompilerLesson/exemples.zip



المراجع

1. THOMAS NIEMANN. A GUIDE TO LEX & YACC
2. AnthonyA.Aaby[2005]. Compiler Construction using Flex and Bison